

Arquitectura de Software

Diseñar para las fallas

Timeout, retry, circuit breaker, bulkhead y load shedding · Logs, métricas y trazas · SLO y error budget

Gerson Garrido

Semana 14

UNIVERSIDAD NACIONAL DE INGENIERÍA

Agenda

Sesión A: resiliencia

1. Fallar es lo normal
2. Lento es peor que caído
3. Timeout
4. Retry
5. Circuit breaker
6. Bulkhead y load shedding

Sesión B: observabilidad

7. Logs, métricas y trazas
8. SLI, SLO y error budget
9. Alertas y health checks

Lectura

A: Nygard, *Release It!* (2.ª ed.), cap. 4 y 5; *Site Reliability Engineering*, cap. 21 y 22; Brooker, *Exponential Backoff And Jitter* (AWS, 2015).
B: *Site Reliability Engineering*, cap. 4 y 6; OpenTelemetry, *Signals*.

SECCIÓN 01

Fallar es lo normal

01

Faltas y fallas

Falta *(fault)*

Algo que sale mal dentro del sistema: un proceso que muere, un servicio externo que tarda.

Falla *(failure)*

Lo que ve quien usa el sistema cuando este deja de hacer lo que debía: la compra que no se completa.

Resiliencia

Las faltas no se evitan. Se diseña para que no se conviertan en fallas: que el sistema siga dando lo esencial mientras ocurren, y se recupere solo.

Diez componentes en cadena, de 99,9 % cada uno: $A = 0{,}999^{10} \approx 0{,}990$.

Puntos de integración: cada llamada que sale del proceso

Lo que hace la dependencia	Lo que ve quien llama	Se entera en
Rechaza la conexión	Un error de conexión	Milisegundos
Responde con un error (503)	Una respuesta de error	Una respuesta
Corta la conexión a la mitad	Un error, sin saber si se procesó	Hasta el corte
Acepta y responde tarde	Nada, hasta que responde	Lo que tarde
Acepta y no responde nunca	Nada	Nunca

Las fallas lentas son las que hacen daño

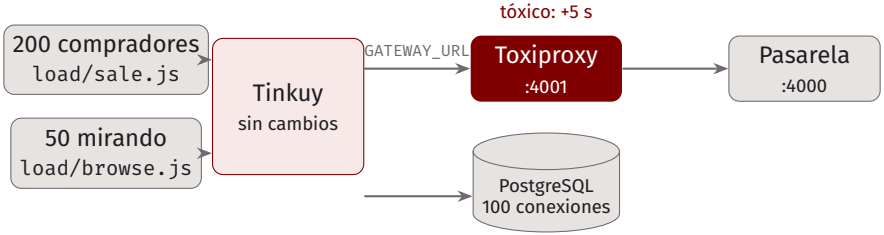
Las tres primeras liberan a quien llama. Las dos últimas lo retienen, si nadie pone un límite.

SECCIÓN 02

Lento es peor que caído

02

Tinkuy, con un proxy entre él y la pasarela



terminal

```
$ make tinkuy-load      # las dos cargas a la vez, 30 s
$ make tinkuy-slow      # toxiproxy-cli toxic add -t latency -a latency=5000
    gateway
```

La pasarela tarda 5 segundos más. ¿Qué le pasa al catálogo?

	Pasarela sana	Pasarela +5 s
Compras con error	1,86 %	75,34 %
Compras pagadas en 30 s	1 360	594
`GET /events` con error	2 de 523	672 de 722
Pedidos del catálogo con error	0,12 %	85,59 %

`GET /events` no usa la pasarela

Y pasó de fallar casi nunca a fallar casi siempre. En cuatro corridas con el tóxico: entre 78 % y 87 % de pedidos del catálogo perdidos.

Tinkuy inicial, en una laptop con Apple Silicon, dentro de Docker. Los números cambian entre corridas; la diferencia, no.

Las conexiones a PostgreSQL, cada dos segundos

make tinkuy-load, con el tóxico

```
   hora   | idle_in_transaction | active | total | max_connections
07:04:44 |                   0 |      1 |     1 | 100
07:04:46 |                  99 |      1 |   100 | 100
07:04:48 |                  99 |      1 |   100 | 100
07:04:50 |                  69 |      2 |    73 | 100
07:04:52 |                  99 |      1 |   100 | 100
```

- `idle in transaction`: una transacción abierta que no ejecuta nada. Espera a la aplicación.
- 99 así, y la número 100 es el `psql` que las cuenta.
- Para todos los demás: `sorry, too many clients already`.

La cadena completa

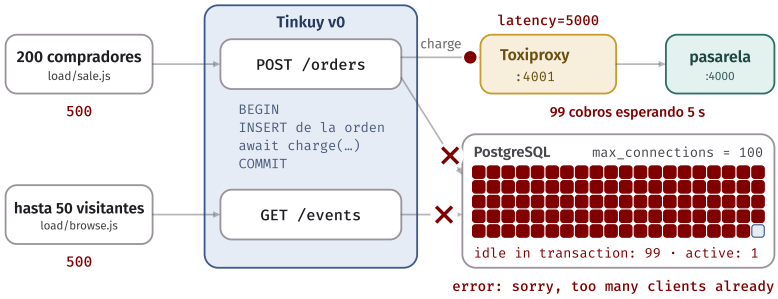
tinkuy/src/orders/create-order.ts

```
1  await client.query('BEGIN');
2  try {
3    const seats = await findSeatsForSale(
       client, input.seatIds);
4    // ... inserta la orden y sus ítems
       ...
5
6    const result = await charge({
7      amountCents: totalCents,
8      reference: `orden-${order.id}`,
9      payment: input.payment,
10   });
11   await insertPayment(client, { /* ...
       */ });
12   // ... entradas, correo ...
13   await client.query('COMMIT');
```

1. La compra abre una conexión, hace `BEGIN` y llama a la pasarela.
2. La pasarela tarda 5 s. La conexión espera, con su transacción abierta. El `fetch` no tiene tiempo máximo.
3. Siguen llegando compras. En dos segundos hay 99 conexiones en ese punto.
4. PostgreSQL acepta 100. La siguiente es rechazada.
5. `GET /events` necesita una conexión. No la hay: 500.

La falla en cascada, paso a paso

TINKUY V0 DETRÁS DE TOXIPROXY, BAJO MAKE TINKUY-LOAD



Una dependencia de una función se puso lenta y cayeron todas las que comparten las conexiones.

SALIDA DE MAKE TINKUY-LOAD	pasarela sana	pasarela 5 s más lenta
orders_paid / orders_error	1360 / 101	594 / 2424
GET /events responde 200: sí / no	521 / 2	50 / 672
conexiones a PostgreSQL	entre 20 y 50 de 100	100 de 100

Por qué lento es peor que caído

Idea clave

Una dependencia caída libera a quien la llama en milisegundos. Una lenta lo retiene, con todo lo que tenía tomado: una conexión, una transacción, memoria.

Con λ pedidos por segundo, y cada uno reteniendo un recurso W segundos, hay en promedio $L = \lambda \cdot W$ recursos ocupados. La dependencia lenta no cambia λ: multiplica W. En Tinkuy, de unos 0,2 s a más de 5 s: por 25.

Falla en cascada *(cascading failure)*

Una falla que crece porque se alimenta a sí misma. Se propaga por el recurso compartido.

Un detalle que engaña

`GET /events`	**Pasarela sana**	**Pasarela +5 s**
Mediana	428,74 ms	2,39 ms
Pedidos con error	2 de 523	672 de 722

La latencia mejoró

Casi todos los pedidos fallan, y fallar es rápido. Quien mire solo la mediana verá un sistema más veloz que nunca.

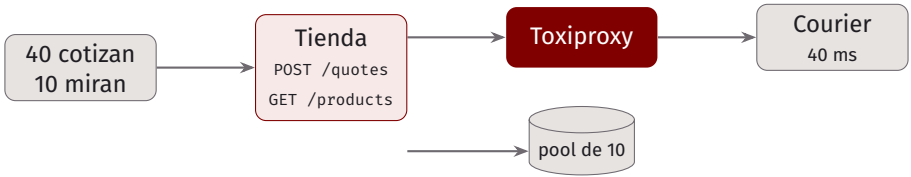
Con un pool de conexiones, el síntoma es el otro: nadie recibe un error, todos **esperan**. Es lo que vamos a ver en el caso de la semana.

SECCIÓN 03

El caso de la semana

03

Chaski: una tienda y su courier




```
make cascade V="healthy none"

== healthy
POST /quotes    201: 1110   503: 0   otros: 0   p50 45 ms   p95 51 ms   máx 342 ms
GET  /products  200: 730   otros: 0   p50 1 ms   p95 16 ms   máx 267 ms
== none
POST /quotes    201: 40   503: 0   otros: 10   p50 15186 ms   p95 19998 ms   máx 19998 ms
GET  /products  200: 20   otros: 0   p50 12550 ms   p95 14979 ms   máx 14985 ms
```

El mismo defecto: llama al courier con una transacción abierta y sin tiempo máximo.

SECCIÓN 04

Timeout

04

Toda llamada remota tiene un tiempo máximo

resilience/timeout.ts

```ts
export async function fetchWithTimeout(url: string, init: RequestInit, timeoutMs: number) {
  try {
    const signal = timeoutMs > 0 ? AbortSignal.timeout(timeoutMs) : undefined;
    return await fetch(url, { ...init, signal });
  } catch (error) {
    if (error instanceof Error && error.name === 'TimeoutError')
      throw new TimeoutError(timeoutMs);
    throw new DependencyError('No se pudo completar la llamada');
  }
}
```

- Sin timeout, la capacidad la decide la dependencia más lenta: sin tope para *W*, no hay tope para *L*.
- Un timeout **no cancela nada** del otro lado *(semana 5)*.

Lo que cambia, medido

```
make cascade V="timeout outside-tx"

== timeout (SHOP_TIMEOUT_MS=1000)
POST /quotes    201: 0   503: 0   otros: 180   p50 3520 ms   p95 4014 ms   máx 4040 ms
GET  /products  200: 59   otros: 0   p50 2810 ms   p95 2828 ms   máx 3788 ms
== outside-tx (SHOP_OUTSIDE_TX=1)
POST /quotes    201: 120   503: 0   otros: 0   p50 5063 ms   p95 5099 ms   máx 5102 ms
GET  /products  200: 750   otros: 0   p50 1 ms   p95 3 ms   máx 38 ms
```

- **Timeout de 1 s**: ninguna cotización sale bien (504), y el catálogo sigue mal: 2,8 s. Acotó *W*; no dejó de retener la conexión.
- **La llamada fuera de la transacción**, sin timeout: el catálogo vuelve a 1 ms. Cortó el camino de la cascada.
- Se necesitan las dos: sin timeout, los pedidos en curso crecen sin límite.

Cómo se elige el valor

Por abajo

La latencia normal de la dependencia, en un percentil alto. Con p99 de 80 ms, un timeout de 100 ms corta llamadas sanas.

Por arriba

Lo que la respuesta todavía vale para quien espera. Y los recursos: una dependencia colgada retiene $\lambda \cdot T$ llamadas.

Error común: subir el timeout “para que no falle”

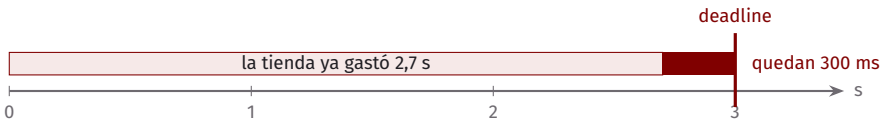
Un error rápido se cambia por uno lento. Timeout de 10 s y courier a 5 s: catálogo con mediana de 10 s, igual que sin timeout.

Trade-off: sacar la llamada de la transacción cuesta atomicidad. Con una lectura no importa; con un cobro hace falta lo de la semana 12.

Deadline propagado

El límite es de la **petición**, no de cada llamada. Se fija una vez y viaja con ella.

$$t_{\text{llamada}} = \text{mín}(t_{\text{propio}},\ d - \text{ahora})$$



Con un timeout propio de 2 s: $\text{mín}(2000,\ 300) = 300$ ms. Si no queda nada, la llamada ni se hace.

En la industria

Site Reliability Engineering, cap. 22: si A eligió un deadline de 30 s y lleva 7 procesando, la llamada a B sale con 23.

SECCIÓN 05

Retry

05



Qué arregla un retry, y cuánto cuesta

make retries

```
1. Toxiproxy corta una de cada tres conexiones (900 llamadas, 30 a la vez)
variante                        bien    mal  intentos  llegaron    tiempo
1 intento                        604    296       900       900    529 ms
3 intentos, sin espera           868     32      1296      1296    644 ms
3 intentos, backoff              868     32      1315      1315   1445 ms
3 intentos, backoff + jitter     864     36      1290      1290   1148 ms
```

Falla solo si fallan los n intentos (3,7 %):

$$p^n = \left(\tfrac{1}{3}\right)^3 = \tfrac{1}{27} \approx 0{,}037$$

Intentos por llamada:

$$1 + p + p^2 = \tfrac{13}{9} \approx 1{,}44$$

Un retry compra confiabilidad con carga: aquí, un 44 % más de pedidos.

Solo sobre operaciones idempotentes

Llegaron = *intentos*: el courier procesó **todos** los pedidos. Se perdió la respuesta.

```
make errors

Sin clave de idempotencia:
  pedidos de la tienda con respuesta   10
  peticiones que recibió el courier    20
  recojos que el courier programó      20
Con clave de idempotencia:
  pedidos de la tienda con respuesta   10
  peticiones que recibió el courier    20
  recojos que el courier programó      10
```

Error común: reintentar una operación que no es idempotente

Con un cobro, el resultado es un cobro doble. Primero la clave de idempotencia; después el retry.

Backoff exponencial y jitter

Cuando la dependencia falla **para todos a la vez**, reintentar de inmediato es llegar dentro de la misma falla.

Backoff exponencial

Esperar cada vez más. Antes del reintento k:

$$\text{mín}(c,\ b \cdot 2^k)$$

Con $b = 200$ ms: 200, 400, 800, 1 600.

Jitter

Un tiempo al azar entre 0 y ese valor:

$$\text{azar}\big(0,\ \text{mín}(c,\ b \cdot 2^k)\big)$$

Cien clientes que fallaron juntos no vuelven juntos.

Brooker (AWS, 2015) llama a esta fórmula *Full Jitter*. Su conclusión: el backoff sin jitter es el claro perdedor.

Una caída de 800 ms y cien clientes a la vez

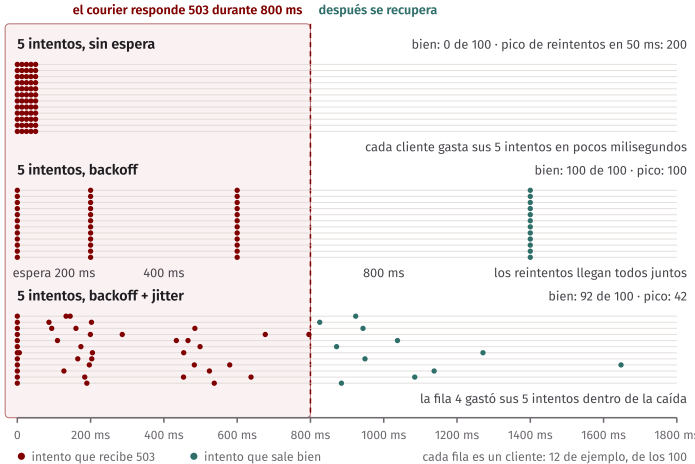
make retries

```
2. El courier responde 503 durante 800 ms; 100 clientes llaman a la vez (base del backoff: 200 ms)
variante                          bien    mal  llegaron  reintentos  pico de reintentos en 50 ms
1 intento                            0    100       100           0                             0
5 intentos, sin espera               0    100       500         400                           200
5 intentos, backoff                100      0       400         300                           100
5 intentos, backoff + jitter        92      8       440         340                            42
```

- **Sin espera**: cinco veces la carga y las mismas fallas. El peor resultado.
- **Backoff**: todas salen bien, pero los reintentos llegan en tres olas de 100.
- **Con jitter**: el pico baja a 42. Ocho clientes gastaron sus intentos dentro de la caída.

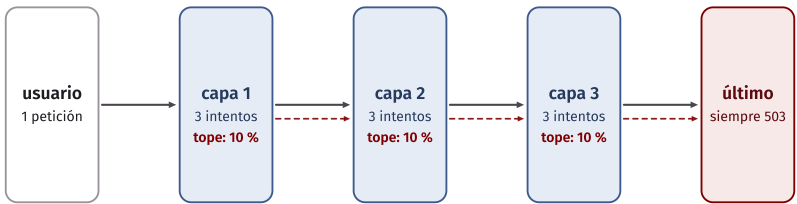
Trade-off: carga repartida para la dependencia, esperas menos predecibles para cada cliente.

Los intentos de cada cliente, en el tiempo



La tormenta de reintentos

1. nadie reintenta | 2. solo la capa 3 | 3. las tres capas | **4. con presupuesto**



La cota es 1,1 × 1,1 × 1,1 = 1,331. Se midieron 130 pedidos por 100 peticiones.

MAKE STORM: PEDIDOS QUE LE LLEGAN AL ÚLTIMO, DE 100 PETICIONES DE USUARIO

nadie reintenta	100 · 1,00 por petición
solo la capa 3 reintenta (3 intentos)	300 · 3,00 por petición
las tres capas, 3 intentos cada una	2 700 · 27,00 por petición
las tres, 3 intentos y presupuesto del 10 %	130 · 1,30 por petición

Cuántos pedidos le llegan al último

Con L capas y n intentos por capa, al último servicio le llegan

$$n^L = 3^3 = 27 \qquad\qquad 4^3 = 64$$

No es una suma: es un producto

El servicio de abajo estaba saturado, y el sistema reaccionó multiplicándole la carga por 27.

Las reglas que la evitan

```
make storm

100 peticiones de usuario; el último servicio falla siempre
quién reintenta                                 le llegan al último  por petición
nadie reintenta                                                 100          1.00
solo la capa 3 reintenta (3 intentos)                           300          3.00
las tres capas, 3 intentos cada una                            2700         27.00
las tres capas, 4 intentos cada una                            6400         64.00
las tres, 3 intentos y presupuesto del 10 %                     130          1.30
```

1. Reintentar en **una sola capa**: la que está pegada a la dependencia.
2. Un **máximo de intentos**, y solo sobre errores que pueden ser transitorios.
3. Un **presupuesto de reintentos**: como mucho, una fracción r de lo que se manda. Con L capas, $(1 + r)^L = 1{,}1^3 = 1{,}331$.

SECCIÓN 06

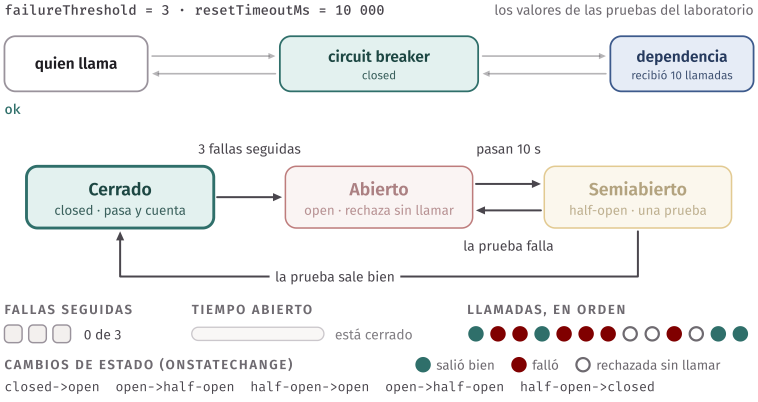
Circuit breaker

06

Tres estados

- **Cerrado**: las llamadas pasan, y se cuenta cómo salen.
- **Abierto**: fallan de inmediato, sin tocar la dependencia.
- **Semiabierto**: pasa *una* llamada de prueba; las demás se rechazan.

Los tres estados, con 3 fallas y 10 s de espera



Cerrado otra vez: vuelve a dejar pasar todo.

Valores de las pruebas del laboratorio. La corrida de `make breaker` que sigue usa 5 fallas seguidas y 3 s.

El circuito, llamada por llamada

```
make breaker

tiempo   circuito   resultado                    duración
2. El courier tarda 5 s más en responder
  1.5 s  closed     timeout                      1001 ms
  ...
  4.8 s  closed     timeout                      1002 ms
  6.0 s  -- el circuito pasa de closed a open
  6.0 s  closed     timeout                      1001 ms
  6.1 s  open       rechazada por el circuito     0.2 ms
3. El courier vuelve a responder
  6.4 s  open       rechazada por el circuito     0.0 ms
  ...
  8.9 s  open       rechazada por el circuito     0.7 ms
  9.4 s  -- el circuito pasa de open a half-open
  9.4 s  -- el circuito pasa de half-open a closed
  9.4 s  half-open  tarifa recibida               47 ms
4. El courier rechaza la conexión
  9.9 s  closed     error de conexión             1.3 ms
  ...
 10.4 s  open       rechazada por el circuito     0.1 ms
El courier recibió 9 pedidos de los 26 que se intentaron.
```

- Lento: de 1 000 ms a 0,1 ms.
- El courier volvió a los 6,4 s; el circuito se enteró a los 9,4. Es el costo.
- Caído: de 1,5 ms a 0,1 ms. Ya era una falla rápida.

Idea clave

Vale lo que vale la diferencia entre fallar lento y fallar rápido.

Decisiones al diseñar un circuit breaker

Decisión	En el laboratorio	Alternativa
Qué es una falla	Timeout, conexión, 5xx	Nunca un 4xx
Cómo se cuenta	Fallas seguidas	Proporción en una ventana
Cuánto dura abierto	Un tiempo fijo	Que crezca con backoff
Alcance	Uno por dependencia y por proceso	

```
bulkhead ( circuit breaker ( retry ( timeout ( fetch ) ) ) )
```

En la industria

Fowler, *CircuitBreaker* (2014): no todo error abre el circuito, y todo cambio de estado se registra.

Qué se responde con el circuito abierto

Es una decisión de negocio, no del circuito.

1. **Un error claro**, pronto: 503 con `retry-after`.
2. **Un valor por defecto**: una tarifa plana marcada como estimada.
3. **Aceptar y hacerlo después**: una cola, con todo lo de la semana 13.

> **Trade-off**
>
> Cambia errores lentos por errores rápidos, y algunos de los rápidos no hacían falta. Agrega estado, parámetros y un modo de falla nuevo que diagnosticar. Conviene con mucho tráfico; sobra en una llamada que se hace una vez por hora.

SECCIÓN 07

Bulkhead y load shedding

07

Bulkhead: que agotar un recurso no agote los demás




```
make cascade V=bulkhead

== bulkhead (SHOP_MAX_CONCURRENT=5)
POST /quotes    201: 15   503: 1010   otros: 0   p50 2 ms   p95 150 ms   máx 5292 ms
GET  /products  200: 716   otros: 0   p50 1 ms   p95 57 ms   máx 170 ms
```

- Sin timeout y con la llamada dentro de la transacción: el catálogo está sano.
- El número sale de $L = \lambda \cdot W$: $74 \cdot 0{,}045 \approx 3{,}3$ llamadas en curso con el courier sano.
- **Trade-off**: la capacidad reservada no se presta.

Load shedding: rechazar pronto para atender bien

Cuando llega más de lo que se puede atender:

1. Aceptarlo todo: la cola crece, y se trabaja para respuestas que nadie lee.
2. Rechazar una parte de inmediato: un 503 en 2 ms.
3. Caerse.

	Qué limita	Quién decide
Load shedding	La carga total que se acepta	El servicio, según su estado
Rate limiting	Lo que pide cada cliente	El servicio, por una regla fija
Backpressure	El ritmo de quien produce	El consumidor le avisa al productor

Error común

Una cola sin límite no es ninguna de las tres: esconde la sobrecarga hasta que se acaba la memoria.

Todas las defensas, bajo la misma carga

	Cotizaciones		Catálogo	
Variante (courier +5 s)	201 / 503 / otros	p50	atendidos	p50
Courier sano	1 110 / 0 / 0	45 ms	730	1 ms
Sin defensas	40 / 0 / 10	15 186 ms	20	12 550 ms
Timeout de 1 s	0 / 0 / 180	3 520 ms	59	2 810 ms
Bulkhead de 5	15 / 1 010 / 0	2 ms	716	1 ms
Fuera de la transacción	120 / 0 / 0	5 063 ms	750	1 ms
Todo junto, con circuit breaker	0 / 1 179 / 7	1 ms	737	1 ms

Idea clave

Ninguna defensa hace que las cotizaciones funcionen: el courier sigue lento. Hacen que fallen rápido, que fallen solas y que el catálogo no se entere.

Degradar: decidir antes qué se apaga primero

```
make cascade V=degrade

== degrade (... SHOP_BREAKER=1 SHOP_FALLBACK=1)
POST /quotes    201: 1186   503: 0   otros: 0   p50 3 ms   p95 65 ms   máx 1039 ms
GET  /products  200: 734    otros: 0   p50 1 ms   p95 18 ms   máx 83 ms
```

- Con el circuito abierto, una tarifa plana con `estimated: true`. Ninguna es la real.
- Si es aceptable lo decide el negocio. La opción tiene que existir, estar marcada y poder contarse.

En Tinkuy

Sin reportes se puede vender. Sin el correo, también. Sin pagos, se puede mostrar el catálogo. Decidirlo y demostrarlo es parte de P5.

Comprobar que las defensas funcionan

terminal

```
$ bash proxy.sh courier 4101 4100
$ toxiproxy-cli toxic add -t latency -a latency=5000 courier
$ toxiproxy-cli toxic add -t reset_peer --tox 0.33 -a timeout=0 courier
$ toxiproxy-cli toggle courier
```

- Una defensa que nunca se ejerció es código sin probar.
- Los tóxicos se aplican **por conexión**, y `fetch` las reutiliza.
- Primero la hipótesis, como un escenario de calidad: “con la pasarela 5 s más lenta, el catálogo responde el 99 % de sus pedidos en menos de 300 ms”. Después el tóxico.

SECCIÓN 08

Logs, métricas y trazas

08

Monitorear y observar

Monitorear

Vigilar las preguntas que ya se conocen: ¿está arriba?, ¿cuántos errores hay?

Observabilidad

Poder entender qué le pasa al sistema mirando lo que emite, también lo que nadie previó.

Señal	Qué pregunta responde	Qué cuesta
Logs	¿Qué pasó en *este* caso?	Crece con el tráfico
Métricas	¿Cuánto, y con qué tendencia?	Poco: según cuántas series hay
Trazas	¿Dónde se fue el tiempo?	Mucho; por eso se muestrean

Tiene consecuencias de arquitectura: cada llamada y cada mensaje llevan contexto.

Logs: de frases a campos

Tinkuy, hoy

```
pasarela: cobro ch_00000001 approved para orden-1304
POST /orders 201
```

la tienda: una línea JSON por pedido


```
{"ts":"2026-11-16T07:06:48.536Z","level":"info","service":"shop","msg":"request",
 "request_id":"8178a60f-4675-43f7-9402-1f049aa4ba34","trace_id":"9dcd3a0e039e1c3db0d18876f9b2e364",
 "span_id":"b7c89f72710a8233","route":"POST /quotes","status":201,"duration_ms":65,"outcome":"ok",
 "courier_attempts":1}
```


- Cuándo, quién, qué pedido, cómo terminó, cuánto tardó.
- El **identificador de la petición**: se genera una vez, va en cada línea y en cada llamada.
- Una línea por pedido, al terminar, con todo. (Aquí, partida en cuatro para que quepa.)

Con campos, las preguntas se responden solas

después de la variante all

```
$ jq -r 'select(.msg == "request" and .route == "POST /quotes") | .outcome' build/logs/all.jsonl | sort | uniq -c
   1109 circuit_open
     70 shed
      7 timeout
$ jq -c 'select(.msg == "breaker_state_change") | {ts, from, to}' build/logs/all.jsonl | head -4
{"ts":"2026-11-16T07:02:42.312Z","from":"closed","to":"open"}
{"ts":"2026-11-16T07:02:47.319Z","from":"open","to":"half-open"}
{"ts":"2026-11-16T07:02:48.319Z","from":"half-open","to":"open"}
{"ts":"2026-11-16T07:02:53.342Z","from":"open","to":"half-open"}
```

- De 1 179 respuestas 503: 1 109 del circuito y 70 del bulkhead. El circuito prueba cada 5 s.

Error común: datos personales o de tarjetas en el log

Un log se copia a muchos lugares y casi nunca se borra. Identificadores, no datos.

Métricas: contadores, medidores e histogramas

```
curl -s localhost:3100/metrics
```

```
http_requests_total{route="POST /quotes",status="201"} 1
http_requests_total{route="POST /quotes",status="503"} 2
http_requests_total{route="POST /quotes",status="504"} 4
db_pool_in_use 0
db_pool_waiting 0
courier_in_flight 0
http_request_duration_ms_bucket{route="POST /quotes",le="25"} 3
...
http_request_duration_ms_bucket{route="POST /quotes",le="1000"} 4
http_request_duration_ms_bucket{route="POST /quotes",le="2500"} 8
...
http_request_duration_ms_count{route="POST /quotes"} 8
```

- **Contador**: solo sube. **Medidor** *(gauge)*: sube y baja. **Histograma**: cuenta por rangos.
- **Cardinalidad**: cada combinación de etiquetas es una serie. Una etiqueta con el id de usuario crea una serie por usuario.

Las cuatro señales doradas

Señal	Qué mide	En la tienda
Latencia	Cuánto tarda un pedido	`http_request_duration_ms`
Tráfico	Cuánta demanda recibe	El ritmo de `http_requests_total`
Errores	Qué fracción falla	Los 5xx, sobre el total
Saturación	Qué tan lleno está lo más escaso	`db_pool_waiting`, `courier_in_flight`

Con el courier sano: `db_pool_in_use 8`, `db_pool_waiting 0`. Con el courier lento: `db_pool_in_use 10`, `db_pool_waiting 40`.

En la industria

Site Reliability Engineering, cap. 6: la latencia de los pedidos exitosos se distingue de la de los fallidos. Y si el compromiso es responder en un segundo, todo pedido más lento cuenta como error.

Trazas y propagación del contexto

`traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01`

versión · `trace-id`: el mismo en todos los servicios · `parent-id`: el span de quien llama · flags

(W3C Trace Context)

Span	Servicio	Operación	Duración	Padre
`b7c89f72710a8233`	shop	`POST /quotes`	65 ms	(raíz)
`d7196c3c7fee1d0c`	courier	`GET /rates`	43 ms	`b7c89f72710a8233`

- Si un servicio no propaga el contexto, la traza se corta ahí.
- Con mensajería, el identificador viaja **en el mensaje**.

La traza de una cotización

UNA COTIZACIÓN, DOS SERVICIOS, UNA TRAZA

tienda (shop)
`POST /quotes · 201 · 65 ms`
`span_id b7c89f72710a8233`
`sin padre: es la raíz`

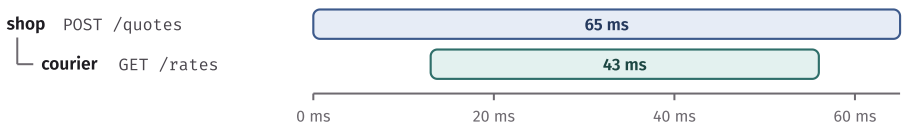
`GET /rates`
con dos cabeceras

courier
`GET /rates · 200 · 43 ms`
`span_id d7196c3c7fee1d0c`
`parent_span_id b7c89f72710a8233`

```
x-request-id: 8178a60f-4675-43f7-9402-1f049aa4ba34
traceparent: 00-9dcd3a0e039e1c3db0d18876f9b2e364-b7c89f72710a8233-01
```

trace-id: igual en los dos — parent-id: el span de la tienda

LA TRAZA, COMO CASCADA DE BARRAS — `trace_id 9dcd3a0e039e1c3db0d18876f9b2e364`



SECCIÓN 09

SLI, SLO y error budget

09



SLI: eventos buenos sobre eventos válidos

$$\text{SLI} = \frac{\text{eventos buenos}}{\text{eventos válidos}}$$

Válido

Un evento que el servicio debía atender. Un 422 por un pedido mal formado no cuenta.

Bueno

Respondió sin error **y a tiempo**. Un 201 a los 5 segundos no es bueno.

- **SLO**: el objetivo sobre el SLI en una ventana. 99 % en 28 días.
- **SLA**: un SLO con una consecuencia pactada si no se cumple.
- Un SLO es un escenario de calidad *(semana 2)*, vigilado en producción.

El error budget

Lo que el SLO permite fallar, sobre los N eventos válidos de la ventana:

$$\text{error budget} = (1 - \text{SLO}) \cdot N$$

Con 99,5 % y un millón de compras al mes:

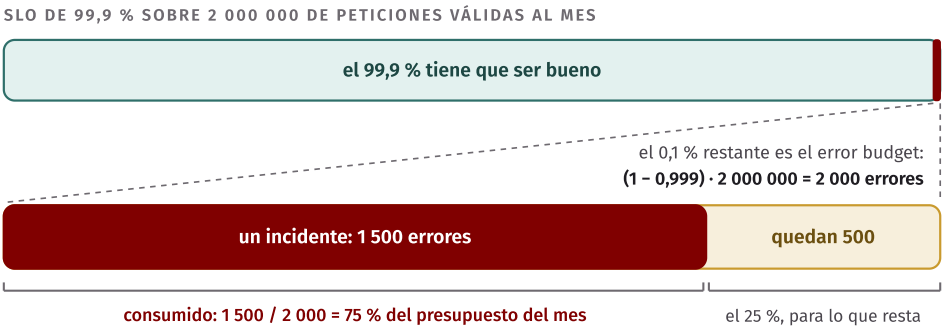
$$(1 - 0{,}995) \cdot 1\,000\,000 = 0{,}005 \cdot 1\,000\,000 = 5\,000 \text{ compras}$$

Con 99,9 % y 2 millones de peticiones, un incidente de 1 500 errores gasta el 75 %:

$$(1 - 0{,}999) \cdot 2\,000\,000 = 2\,000 \qquad \frac{1\,500}{2\,000} = 0{,}75$$

The Site Reliability Workbook, cap. 2: con 99,9 % y 3 millones de peticiones en cuatro semanas, el presupuesto es de 3 000 errores; un incidente de 1 500 cuesta el 50 %.

Una franja mínima del total



Para qué sirve un presupuesto

SLO	En 28 días
99 %	403,2 min
99,5 %	201,6 min
99,9 %	40,3 min
99,99 %	4,0 min

$28 \cdot 24 \cdot 60 = 40\,320$ minutos

- **Mientras queda**: se despliega y se experimenta. Está para gastarse.
- **Cuando se acaba**: manda la confiabilidad.
- Convierte “¿qué tan confiable?” en un número que negocian producto y operación.

100 % es el objetivo equivocado

Deja un presupuesto de cero, cada nueve cuesta más que el anterior, y el usuario no nota la diferencia: su propia red falla más.

Del log al SLI

```
make sli V=all

SLI: POST /quotes que respondieron bien en menos de 1000 ms
  peticiones a la ruta   1186
  no válidas (4xx)       0
  válidas                1186
  buenas                 0
  malas                  1186 (1186 con error 5xx, 0 lentas)
  SLI                    0 / 1186 = 0.00 %
SLO de 99 % sobre una ventana de 100000 peticiones válidas
  error budget           1000 peticiones malas
  consumido aquí         1186 / 1000 = 118.60 %
```

Variante	SLI	Malas	Gastó
Courier sano	100,00 %	0	0,0 %
Sin defensas	0,00 %	60	6,0 %
Bulkhead	0,00 %	1 025	102,5 %
Todo junto	0,00 %	1 186	118,6 %
Degradando	99,41 %	7	0,7 %

Error común: comparar el presupuesto entre experimentos de modelo cerrado

La variante sin defensas no tuvo menos fallas: tuvo menos **pedidos**, porque frenó a quien la medía. Entre experimentos se compara el SLI.

SECCIÓN 10

Alertas y health checks

10



Alertar por síntomas, no por causas

Síntoma

Lo que el usuario nota: las compras fallan, el catálogo tarda.

Causa

Lo que puede producirlo: el CPU al 90 %, el pool lleno, la pasarela lenta.

- Las causas no se pueden enumerar: nadie habría puesto una alerta para la cascada de hoy.
- Se alerta cuando el SLI se aleja del SLO. Las causas van a los tableros.
- Cada alerta pide una **acción humana**. Las que no, cansan y terminan ignoradas.
- *Site Reliability Engineering*, cap. 6: si merece una respuesta mecánica, no debería ser una alerta.

Health checks: liveness y readiness

Liveness

¿El proceso está vivo? Si no, la plataforma lo **reinicia**.
En la tienda: `GET /health`.

Readiness

¿Puede atender ahora? Si no, se le **deja de enviar tráfico**.
En la tienda: `GET /ready`.

Error común: un health check que comprueba todas las dependencias

El día que el courier se caiga, **todas** las instancias dirán que no pueden atender, y la caída de una dependencia que solo afecta a las cotizaciones dejará sin servicio al catálogo.

`/ready` comprueba la base de datos de la tienda, no al courier. El estado del circuito se informa; no se usa para sacar la instancia.

Lo que cuesta observar

- Latencia y CPU: cada línea de log y cada span se arman y se envían.
- Almacenamiento: 300 bytes por pedido, a 1 000 pedidos por segundo, son unos 26 GB al día ($300 \cdot 1\,000 \cdot 86\,400 \approx 2{,}6 \cdot 10^{10}$ bytes).
- **Trade-off**: más detalle es más preguntas que se pueden responder después, y más costo todos los días. Lo habitual: métricas de todo, logs con retención corta, trazas muestreadas.

En Tinkuy

Hoy: frases sueltas, sin hora, sin identificador, sin duración. P5 pide logs estructurados, las métricas del SLO de compra y la traza de una compra a través del monolito, el broker y el servicio de pagos.

Laboratorio: una pasarela de pagos lenta y caída

En parejas, sobre su copia de Tinkuy, con `code/week-14/`.

1. `svc start toxiproxy`, `bash proxy.sh gateway 4001 4000` y `GATEWAY_URL`.
2. Línea de base: las dos cargas a la vez.
3. Tóxico de 5 s. Abrir `psql` **antes**. Explicar la cadena completa.
4. Timeout con `AbortSignal.timeout`, y la espera fuera de la transacción.
5. Retry, sin espera y con backoff y jitter, cortando una de cada tres conexiones.
6. Completar `lab/circuit-breaker.ts` (`make lab`): pasarela caída y pasarela lenta.
7. Bulkhead de 20 y 503 inmediato. ¿Por qué 20?
8. Una línea JSON por compra; SLI y error budget de cada experimento.

Para la próxima clase

- Esta semana se entrega **P4 · Pagos como servicio** y se publica **P5 · Cuando algo falla**.
- Guardar el código de cada paso del laboratorio: es el punto de partida de P5.
- Semana 15, evolución. Leer *Software Architecture in Practice*, cap. 23 *Managing Architecture Debt*, y *Building Evolutionary Architectures* (2.ª ed.), cap. 2 *Fitness Functions* (Sesión A).
- Lecturas libres: Fowler, *StranglerFigApplication*; Conway, *How Do Committees Invent?* (1968).

Pregunta

Con la pasarela 5 segundos más lenta, ¿qué debería seguir funcionando en Tinkuy? Escríbanlo como un escenario, con su medida, antes de tocar el código.

Lecturas de hoy: Nygard, *Release It!* (2.ª ed.), cap. 4 y 5; *Site Reliability Engineering*, cap. 4, 6, 21 y 22; Brooker, *Exponential Backoff And Jitter*; OpenTelemetry, *Signals*.