

Arquitectura de Software

Tácticas

Modificabilidad · Rendimiento · Disponibilidad · Seguridad

Gerson Garrido

Semana 6

UNIVERSIDAD NACIONAL DE INGENIERÍA

Agenda

Sesión A

1. Qué es una táctica
2. Tácticas de modificabilidad
3. Rendimiento: a dónde se va el tiempo
4. El pool de conexiones
5. N+1 y cache

Sesión B

6. Disponibilidad: falta, error y falla
7. Degradar en vez de caer
8. El retry
9. Seguridad
10. Una táctica cuesta otro atributo

Lectura

A: *Software Architecture in Practice*, cap. 8 *Modifiability* y cap. 9 *Performance*.
B: *Software Architecture in Practice*, cap. 4 *Availability* y cap. 11 *Security*; Saltzer y Schroeder, *The Protection of Information in Computer Systems* (1975), principios de diseño.

SECCIÓN 01

Qué es una táctica

01

De las decisiones grandes a las pequeñas

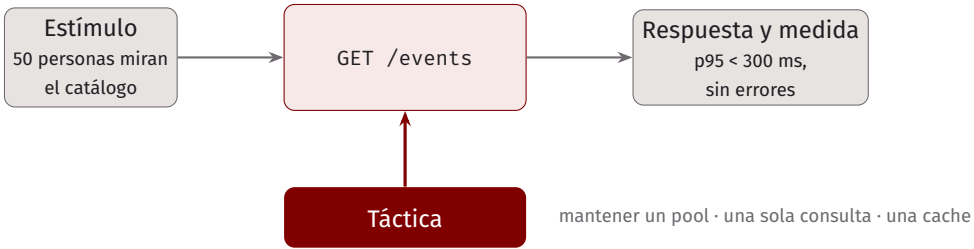
Semanas 4 y 5: **estilos**. Decisiones grandes, que se toman pocas veces.

Tinkuy va a seguir siendo un monolito varias entregas. Y aun así hay que decidir:

- ¿Cómo consigue sus conexiones a la base de datos?
- ¿Cuántas consultas hace para armar una lista?
- ¿Qué pasa si el servidor de correo no responde?
- ¿Quién puede hablar con la pasarela?

> **Táctica** *(tactic)*
>
> Una decisión de diseño que influye sobre la respuesta a un estímulo de **un** atributo de calidad.

Una decisión que mueve una medida



1. **El escenario**, con su medida.
2. **Medir** el sistema como está y encontrar dónde se pierde la medida.
3. **Elegir la táctica** que corresponde a esa causa, mirando lo que cuesta.
4. **Medir de nuevo**: mismo script, misma máquina.

Táctica, patrón y estilo

	Qué es	Ejemplo
Táctica	Una decisión, para un atributo	Restringir dependencias; mantener un pool; heartbeat
Patrón	Varias tácticas que se toman juntas	Microkernel; réplicas tras un load balancer
Estilo	La forma del sistema completo	Modular monolith; microservicios

Microkernel =

encapsular + usar un intermediario (el registro) + diferir el enlace

Load balancer con health checks =

copias del cómputo *(rendimiento)* + ping y eco + sacar de servicio *(disponibilidad)*

Un solo atributo; lo demás es el trade-off

- Un pool de conexiones es una táctica de **rendimiento**.
- Su efecto sobre los demás atributos no viene en el nombre: hay que buscarlo.
- El libro agrupa las tácticas de cada atributo en un **árbol**, y trae un **cuestionario** para revisar un diseño.
- Un catálogo es un inventario, no una lista de tareas.

Idea clave

"Usamos Redis" no dice nada de la arquitectura.

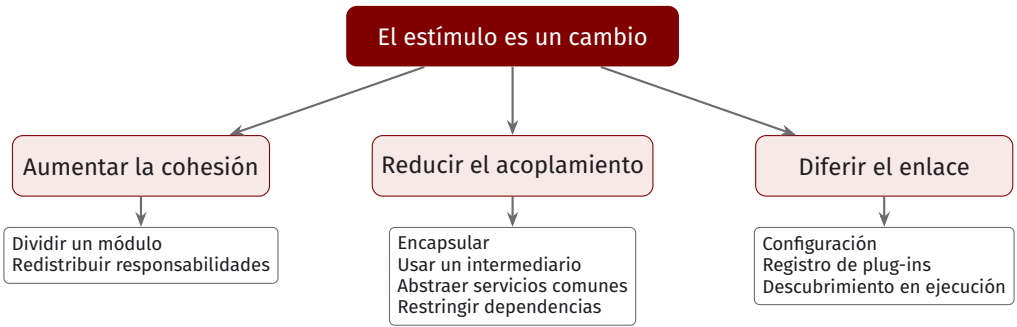
"Mantenemos una copia de los datos de lectura, con un TTL de 5 segundos" dice qué se gana y qué se arriesga.

SECCIÓN 02

Tácticas de modificabilidad

02

Modificabilidad: el árbol ya lo conocen



Semanas 3 y 4, con nombre: KWIC, ocultamiento de información, ports y adapters, `depcruise`, microkernel.

Diferir el enlace: decidir más tarde

Momento	Mecanismo	Cambiar el valor exige
Al escribir el código	Una constante, un `import` directo	Editar, probar y desplegar
Al desplegar	Un archivo de configuración empaquetado	Volver a desplegar
Al arrancar	Variables de entorno, registro de plug-ins	Reiniciar el proceso
En ejecución	Feature flags, descubrimiento de servicios	Nada

Trade-off

Cuanto más tarde se enlaza, más barato es el cambio y más cara la infraestructura que lo permite. Cada opción multiplica las configuraciones que hay que probar.

El servicio de la semana se configura así

src/variant.ts

```ts
export type Variant = {
  /** connect: una conexión nueva por consulta. pool: un pg.Pool. */
  db: 'connect' | 'pool';
  poolSize: number;
  /** nplus1: una consulta por título. single: una sola consulta con GROUP BY. */
  list: 'nplus1' | 'single';
  cache: 'off' | 'redis';
  cacheTtlSeconds: number;
  /** Qué hacer si Redis no responde: fallar o consultar la base de datos. */
  onCacheError: 'fail' | 'degrade';
};
```

Probar un pool de 50 no es un cambio de código: `POOL_SIZE=50 make serve`. Antes hubo que **encapsular** la capa de datos.

SECCIÓN 03

Rendimiento: a dónde se va el tiempo

03

El estímulo y la medida

Estímulo: la llegada de eventos.

- **Periódica**: un reporte cada minuto.
- **Esporádica**: gente mirando el catálogo.
- **En ráfaga**: la apertura de ventas, a las 10:00:00.

Medida de la respuesta:

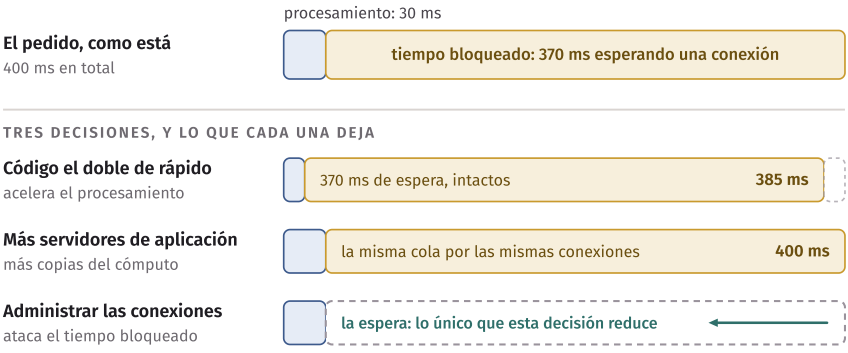
- Latencia, con percentiles.
- Throughput.
- Tasa de pedidos perdidos.

El tiempo de un pedido

$$t_{\text{respuesta}} = t_{\text{procesamiento}} + t_{\text{bloqueado}}$$

Bloqueado: esperando un recurso (una conexión, un lock, un procesador) o a otro componente.

400 ms: 30 de procesamiento, 370 esperando una conexión

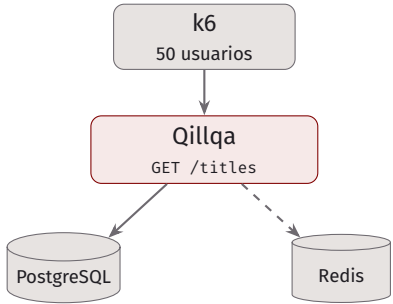


Tres decisiones contra el mismo pedido

- Hacer el código el doble de rápido ahorra **15 ms de 400**.
- Más servidores de aplicación: los nuevos hacen la misma cola por las mismas conexiones. Si cada uno abre las suyas, la base de datos queda peor.
- Lo que hay que atacar es la espera: **administrar ese recurso**.

> **Idea clave**
>
> Primero se mide dónde se va el tiempo; después se elige la táctica.

El caso de la semana: Qillqa



El catálogo de una red de bibliotecas, en `code/week-06/`:

- 4 sedes, 31 títulos, 11 605 ejemplares.
- `GET /titles`: la lista, con los ejemplares disponibles de cada título.
- Se parece a la lista de eventos de Tinkuy **a propósito**.

Escenario

50 personas consultan el catálogo a la vez. `GET /titles` responde con un p95 menor de 100 ms y sin errores.

¿La consulta es lenta?

```
psql -d arq_w06

 Aggregate  (cost=14.47..14.48 rows=1 width=4) (actual time=0.036..0.036 rows=1.00 loops=1)
   ->  Index Scan using copies_title_idx on copies  (cost=0.29..14.03 rows=178 width=0) (actual time=0.009..0.029 rows=178.00 loops=1)
...
 Execution Time: 0.063 ms
```

```
después de un solo pedido a GET /titles

{
  "queries": 32,
  "connectionsOpened": 32
}
```

La consulta tarda cerca de un milisegundo. El pedido la hace **31 veces**, más una para la lista, y abre **una conexión para cada una**.

Dos causas distintas

1. Cada consulta paga una conexión nueva

Cómo se administra un recurso.

Táctica: mantener un pool.

2. 32 consultas donde bastaría una

Cuánto trabajo pide cada pedido.

Táctica: reducir el costo del pedido.

- **En Tinkuy**: `src/db.ts` abre una conexión por consulta, y `GET /events` hace un `COUNT` por evento. Las mismas dos causas.

Error común: optimizar sin medir

Un índice para una consulta que tarda un milisegundo no mueve el p95. La táctica correcta para el cuello de botella equivocado no cambia nada.

Tácticas de rendimiento: controlar la demanda

Táctica	Qué se hace
Limitar la tasa de llegada	Rate limiting: el exceso recibe un rechazo rápido
Limitar la respuesta a eventos	Cola de tamaño máximo; con la cola llena se descarta
Priorizar	La compra antes que el reporte
Reducir el costo de cada pedido	Un algoritmo mejor, menos idas y vueltas a la base de datos
Acotar el tiempo de ejecución	El pedido que pasa de su límite se corta

Cuatro de las cinco aceptan no atender del todo a alguien para proteger al resto. Reducir el costo es la única que no le quita nada a nadie: se mira primero.

Tácticas de rendimiento: administrar los recursos

Táctica	Qué se hace
Aumentar los recursos	Escalar hacia arriba o hacia los lados
Introducir concurrencia	Atender varios pedidos a la vez
Varias copias del cómputo	Instancias detrás de un load balancer
Varias copias de los datos	Cache, réplicas de lectura
Acotar el tamaño de las colas	Toda cola tiene un máximo
Planificar el uso de un recurso	Una política decide quién va primero

El pool no está en la tabla

Es un mecanismo que aplica tres: reutilizar *(reduce el costo)*, un máximo *(acota)*, y una cola por orden de llegada *(planifica)*.

SECCIÓN 04

El pool de conexiones

04



Lo que cuesta una conexión

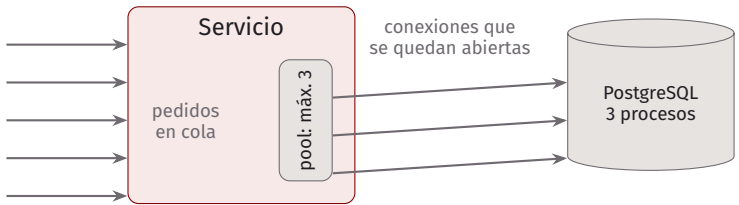
- Una negociación (protocolo, autenticación, parámetros; cifrado si va por la red) y un **proceso** en el servidor por cada cliente.
- En el contenedor es de las más baratas: socket local, sin contraseña.

```
make connect-cost

200 consultas, una después de otra (host: /var/run/postgresql)
conectar + consulta + cerrar          p50   2.18 ms   p95   2.87 ms
consulta sobre una conexión abierta   p50   0.13 ms   p95   0.18 ms
razón entre las medianas: 16.6
```

Y eso es una detrás de otra. Con cincuenta a la vez, el servidor crea y destruye decenas de procesos por segundo mientras intenta responder consultas.

Un pool: reutilizar y poner un tope



- Quien necesita una consulta toma una conexión libre, la usa y la devuelve.
- Si todas están ocupadas, **espera su turno dentro del servicio**. PostgreSQL no se entera.

Con pool y sin pool, la misma carga

```
make measure V="baseline pool-10"

k6: 50 usuarios virtuales durante 15s, GET /titles
variante                                  p50        p95        máx  pedidos   errores conexiones consultas
Sin pool, N+1 (línea de base)          571.4 ms   721.2 ms   985.7 ms     78/s    0.00 %         34      32.0
Pool de 10, N+1                         23.5 ms    43.3 ms    78.7 ms    664/s    0.00 %         10      32.0
```

- El mismo script, las mismas 32 consultas por pedido. Lo que cambió: ya no se abre una conexión para cada una.
- La columna **conexiones**: con `max: 10`, PostgreSQL nunca ve más de diez.

El tope: 200 usuarios en vez de 50

```
VUS=200 make measure V="baseline pool-10"

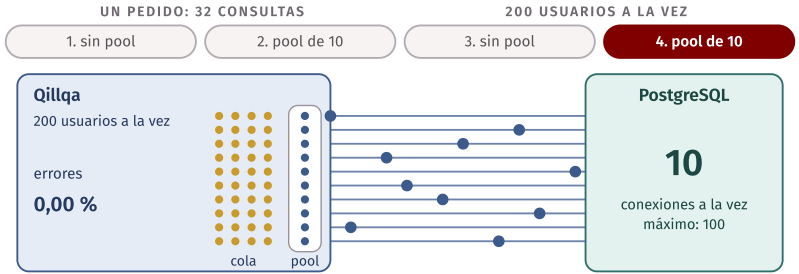
k6: 200 usuarios virtuales durante 15s, GET /titles
variante                                  p50        p95        máx  pedidos   errores conexiones consultas
Sin pool, N+1 (línea de base)          325.5 ms  1589.2 ms  3039.7 ms    351/s   97.76 %         98       6.0
Pool de 10, N+1                        107.1 ms   133.5 ms   201.9 ms   1256/s    0.00 %         10      32.0
```

- PostgreSQL acepta 100 conexiones (`max_connections`). Sin pool, nada impide pedirle más: `sorry, too many clients already`.
- Con pool, los 200 usuarios se atienden con diez conexiones y **sin errores**.

Idea clave

Un pool hace dos cosas: ahorra el costo de conectar, y protege a la base de datos de su propia aplicación.

Las dos cosas que hace un pool



PostgreSQL nunca ve más de 10 conexiones, y no hay un solo error.

MAKE MEASURE, CON 50 Y CON 200 USUARIOS VIRTUALES

carga	variante	p95	errores	conexiones
50 usuarios	Sin pool, N+1	721.2 ms	0.00 %	34
50 usuarios	Pool de 10, N+1	43.3 ms	0.00 %	10
200 usuarios	Sin pool, N+1	1589.2 ms	97.76 %	98
200 usuarios	Pool de 10, N+1	133.5 ms	0.00 %	10

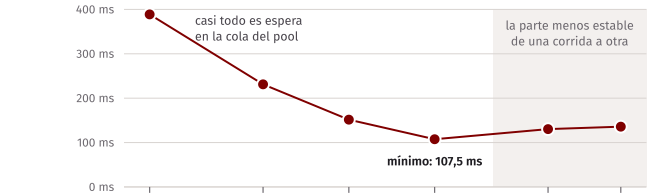
¿Cincuenta conexiones son mejores que diez?

make curve (200 usuarios virtuales)

```
k6: 200 usuarios virtuales durante 15s, GET /titles
variante                              p50        p95        máx  pedidos   errores conexiones consultas
Sin pool, N+1 (línea de base)     424.9 ms  1996.1 ms  4761.2 ms    273/s  97.35 %         95       5.8
Pool de 2, N+1                    343.4 ms   388.8 ms   444.0 ms    502/s   0.00 %          2      32.0
Pool de 5, N+1                    156.1 ms   231.2 ms   288.2 ms    937/s   0.00 %          5      32.0
Pool de 10, N+1                    79.2 ms   151.6 ms   248.7 ms   1457/s   0.00 %         10      32.0
Pool de 20, N+1                    75.2 ms   107.5 ms   195.6 ms   1559/s   0.00 %         20      32.0
Pool de 50, N+1                    84.4 ms   130.3 ms   322.5 ms   1422/s   0.00 %         50      32.0
Pool de 90, N+1                    98.4 ms   135.8 ms   501.5 ms   1296/s   0.00 %         90      32.0
```

La curva del tamaño del pool

P95 DE GET /TITLES, 200 USUARIOS VIRTUALES, UNA CORRIDA DE MAKE CURVE



tamaño del pool	2	5	10	20	50	90
p95 (ms)	388,8	231,2	151,6	**107,5**	130,3	135,8
pedidos/s	502	937	1457	**1559**	1422	1296

- **Muy pequeño**: sin errores, pero casi todo el tiempo es cola. Tiempo bloqueado puro.
- **En el medio**: el mejor punto.
- **Muy grande**: no mejora, y en esta máquina empeora.

Tres de cuatro corridas dieron esta forma; en la otra, de 20 a 90 salió parecido.

Directo a la base de datos, sin servidor HTTP

```
make db-curve

Procesadores que ve el contenedor: 14
conexiones  consultas/s  p50 por consulta  p95 por consulta
         1          781          1.25 ms           1.38 ms
         2         1534          1.28 ms           1.42 ms
         4         3020          1.29 ms           1.50 ms
         8         4178          1.91 ms           2.22 ms
        16         5095          2.62 ms           6.08 ms
        32         5035          5.80 ms          12.24 ms
        64         5201         11.06 ms          25.01 ms
```

- Hasta cierto punto, más conexiones son más consultas por segundo.
- Después el total casi no sube, y **cada consulta tarda más**.
- Una consulta necesita un procesador. La conexión de más no agrega capacidad: agrega un competidor.

Cómo se elige el tamaño

Cuando hay más pedidos que capacidad, alguien espera. La decisión es **dónde**:

- En la cola del pool: barato, y la base de datos trabaja a su mejor ritmo.
- Dentro de la base de datos: procesos que compiten entre sí.
- **El pool es por proceso**: 4 instancias $\times$ 50 = 200 conexiones.
- Una conexión retenida durante una llamada lenta no la usa nadie más.

En la industria

La wiki de PostgreSQL da un punto de partida: cerca de dos conexiones activas por núcleo, más los discos en uso. Y se confirma midiendo.

Error común

Subir el pool cada vez que hay espera por conexiones.

wiki.postgresql.org/wiki/Number_Of_Database_Connections

SECCIÓN 05

N+1 y cache

05



N+1: una consulta por elemento

src/titles.ts

```ts
export async function listTitlesNPlusOne(db: Db, filter: TitleFilter = {}) {
  const rows = await db.query<TitleRow>('SELECT t.id, t.name, ... FROM titles t ...',
    [filter.city ?? null]);
  const titles: TitleSummary[] = [];
  for (const row of rows) {
    titles.push(toSummary(row, await countAvailableCopies(db, row.id)));
  }
  return titles;
}
```

- Una consulta para N elementos y N más, una por elemento: $N + 1$ viajes a la base de datos.
- Nace al **componer código limpio**: el problema es el `for` que junta dos funciones correctas.

Una sola consulta

src/titles.ts (la consulta de listTitlesSingleQuery)

```sql
SELECT t.id, t.name, t.author, t.year, b.name AS branch_name, b.city AS branch_city,
       count(c.id) FILTER (WHERE c.status = 'available')::int AS available
  FROM titles t
  JOIN branches b ON b.id = t.branch_id
  LEFT JOIN copies c ON c.title_id = t.id
 WHERE ($1::text IS NULL OR b.city = $1)
 GROUP BY t.id, b.id
 ORDER BY t.name, t.id
```

- `LEFT JOIN`: conserva los títulos sin ningún ejemplar.
- `FILTER`: cuenta los disponibles sin descartar el título con todo prestado.
- La versión con `JOIN` y `WHERE c.status = 'available'` no da error: devuelve 30 títulos en vez de 31.

Antes de medir: ¿responde lo mismo?

make test (fragmento)

```
✓ todas las variantes responden lo mismo a GET /titles (3105.092334ms)
✓ todas las variantes responden lo mismo a GET /titles?city=Cusco (665.395792ms)
✓ todas las variantes responden lo mismo a GET /titles?city=Lima (1477.351668ms)
✓ todas las variantes responden lo mismo a GET /titles?city=Puno (131.067917ms)
✓ todas las variantes responden lo mismo a GET /titles/7 (381.340625ms)
✓ la lista tiene los 31 títulos, también el que no tiene ejemplares (740.148376ms)
✓ N+1 hace 32 consultas por pedido (1 + 31 títulos) y la versión de una consulta hace 1 (39.36625ms)
✓ errors/list-inner-join.ts no es equivalente: pierde el título sin ejemplares (4.212042ms)
```

Idea clave

Una táctica de rendimiento no debe cambiar lo que el sistema responde. Primero la equivalencia, con los casos de borde; después la medición.

Dos causas, dos tácticas, cuatro combinaciones

```
make measure V="baseline pool-10 single-nopool single"

k6: 50 usuarios virtuales durante 15s, GET /titles
variante                              p50        p95        máx  pedidos   errores conexiones consultas
Sin pool, N+1 (línea de base)      572.3 ms   651.9 ms   709.6 ms     79/s    0.00 %         38      32.0
Pool de 10, N+1                     30.8 ms    47.1 ms   100.0 ms    611/s    0.00 %         10      32.0
Sin pool, una consulta               8.1 ms    14.1 ms    95.3 ms    832/s    0.00 %         21       1.0
Pool de 10, una consulta             4.7 ms    11.6 ms    38.2 ms    872/s    0.00 %         10       1.0
```

- Cualquiera de las dos, sola, se lleva casi toda la mejora: las dos eliminan las 32 conexiones por pedido.
- Juntas dan el mejor número, y solo el pool pone un tope.
- **Costo de quitar el N+1**: la consulta única sabe de títulos, sedes y ejemplares. Si fueran módulos o servicios distintos, ese `JOIN` cruzaría un límite.

Cache, primera vista

Cache-aside con TTL:

1. Buscar en la cache.
2. Si está: devolverla *(hit)*.
3. Si no *(miss)*: calcular, guardar con un tiempo de vida y devolver.

Nadie invalida: la entrada vale hasta que vence.

src/cache.ts (el núcleo)

```ts
cached = await client.get(fullKey);
if (cached !== null) {
  counters.hits += 1;
  return { body: cached, outcome: 'hit' };
}
counters.misses += 1;
const body = await compute();
await client.set(fullKey, body, { EX: ttl });
return { body, outcome: 'miss' };
```

Lo que se gana

```
make measure V="single cache"

k6: 50 usuarios virtuales durante 15s, GET /titles
variante                                  p50        p95        máx  pedidos   errores conexiones consultas
Pool de 10, una consulta                4.7 ms    10.7 ms    47.7 ms    876/s    0.00 %         10       1.0
Pool de 10, una consulta, cache de 5 s  1.0 ms     2.5 ms    43.0 ms    905/s    0.00 %         10       0.0
```

- Latencia para quien pide.
- Carga sobre la base de datos: la columna **consultas** cae a casi cero. Queda libre para lo que no se puede guardar en una cache.

Lo que se abre: el dato viejo

```
make stale

antes del préstamo    lista: 178 (x-cache: miss)   detalle: 178
{"loan":{"titleId":1,"copyId":1}}
0 s después           lista: 178 (x-cache: hit)   detalle: 177
1 s después           lista: 178 (x-cache: hit)   detalle: 177
2 s después           lista: 178 (x-cache: hit)   detalle: 177
3 s después           lista: 178 (x-cache: hit)   detalle: 177
4 s después           lista: 178 (x-cache: hit)   detalle: 177
5 s después           lista: 177 (x-cache: miss)   detalle: 177
6 s después           lista: 177 (x-cache: hit)   detalle: 177
```

Durante cinco segundos, dos respuestas distintas a la misma pregunta. ¿Es grave? Depende del escenario. Lo que salva al diseño: la lista solo *informa*; la decisión de prestar se toma contra la base de datos. *(Semana 11: invalidación.)*

La cache que esconde una consulta

```
make measure V="cache-nplus1 cache"

k6: 50 usuarios virtuales durante 15s, GET /titles
variante                                  p50        p95        máx  pedidos   errores conexiones consultas
Sin pool, N+1, cache de 5 s             0.9 ms     2.7 ms  1087.0 ms    754/s    0.00 %         30       0.4
Pool de 10, una consulta, cache de 5 s  1.0 ms     2.6 ms    30.7 ms    912/s    0.00 %         10       0.0
```

- Sin pool y con N+1, con la cache encima: el p50 y el p95 se ven bien.
- Miren **máx** y **conexiones**: cada vez que vence el TTL, todos los pedidos que llegan recalculan a la vez, con 32 conexiones cada uno.

Error común

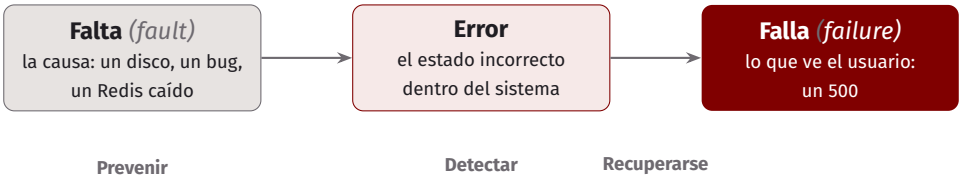
Poner una cache para esconder una consulta mal escrita. La consulta sigue ahí y aparece con cada fallo de cache.

SECCIÓN 06

Disponibilidad

06

Falta, error y falla



Idea clave

Las faltas no se pueden eliminar. La disponibilidad se diseña para que no lleguen a ser fallas, o para que las fallas duren poco.

Detectar

Monitor	Un componente vigila a los demás
Ping y eco	El monitor pregunta
Heartbeat	El componente avisa
Timestamp	Orden y repetidos
Monitoreo de condiciones	Invariantes, sumas
Comprobación de cordura	¿El resultado es razonable?
Votación	Varios calculan; se compara
Detección de excepciones	Error, o tiempo vencido
Autoprueba	El componente se prueba

La más simple

Un **timeout**: pasado el límite, el silencio se trata como un error.

Sin detección no hay recuperación

Tinkuy llama a la pasarela sin timeout: detecta una pasarela *caída*, no una *lenta*.

Recuperarse y prevenir

Preparación y reparación

Redundancia: activa, pasiva o de repuesto
Rollback
Manejo de excepciones
Retry
Ignorar el comportamiento defectuoso
Degradación
Reconfiguración

Reintroducción

Modo sombra
Resincronización del estado
Reinicio escalonado

Prevenir

Sacar de servicio
Transacciones
Modelos predictivos
Prevención de excepciones
Aumentar el conjunto de competencias

Conjunto de competencias: las situaciones en las que un componente sabe operar.

Degradar en vez de caer

Seguir ofreciendo lo esencial y suspender lo demás.

- Si el correo no responde, la compra podría completarse y el correo salir después.
- Si el reporte de ventas no responde, la venta no debería enterarse.
- En Tinkuy v0, el correo va dentro de la transacción: si falla, la compra se deshace.

¿Qué es esencial?

¿Se puede vender sin enviar el correo? ¿Sin poder cobrar?

No lo decide quien programa: es una decisión de **negocio**, y el arquitecto la tiene que preguntar.

Con la cache, `GET /titles` depende de PostgreSQL **y** de Redis: en serie, quedó menos disponible que antes. Y Redis solo tiene una copia.

La cache se cae: fallar

```
make cache-down

CACHE_ON_ERROR=fail: 40 pedidos por segundo a GET /titles
  s   200   500   hit  miss  bypass   p50      suceso
  0    40     0    40     0       0    2.1 ms
  1    40     0    39     1       0    1.8 ms
  2    40     0    40     0       0    1.5 ms
  3     0    40     0     0       0    1.5 ms   svc kill redis
  4     0    40     0     0       0    1.5 ms
  5     0    40     0     0       0    1.5 ms
  6    17    23    16     1       0    2.4 ms   svc start redis
  7    40     0    40     0       0    2.4 ms
  8    40     0    40     0       0    2.2 ms
pedidos: 360, con error: 143 (39.7 %)
```

PostgreSQL estuvo sano todo el tiempo y tenía la respuesta. Una falta en un componente opcional se convirtió en una falla total del endpoint.

La cache se cae: degradar

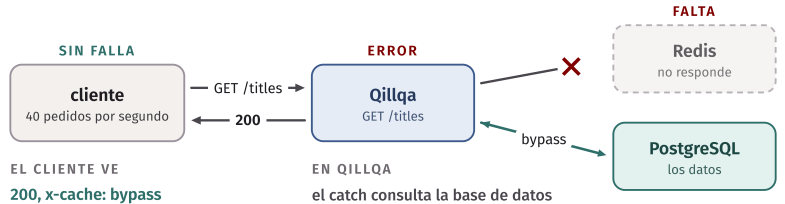
```
make cache-down
```

```
CACHE_ON_ERROR=degrade: 40 pedidos por segundo a GET /titles
  s   200   500   hit  miss  bypass   p50      suceso
  0    40     0    40     0       0    2.4 ms
  1    40     0    39     1       0    2.1 ms
  2    40     0    40     0       0    3.3 ms
  3    40     0     0     0      40   10.0 ms   svc kill redis
  4    40     0     0     0      40    8.7 ms
  5    40     0     0     0      40    8.3 ms
  6    40     0    17     1      22    7.9 ms   svc start redis
  7    40     0    40     0       0    2.1 ms
  8    40     0    40     0       0    2.3 ms
pedidos: 360, con error: 0 (0.0 %)
```

Más lento, no caído. Tres tácticas: detección de excepciones, degradación y un conjunto de competencias mayor.

La misma falta con dos finales

1. Redis arriba | 2. Redis caído: fail | **3. Redis caído: degrade**



El error no llegó a ser una falla: 200 con x-cache: bypass.

MAKE CACHE-DOWN: LOS TRES SEGUNDOS SIN REDIS

variante	pedidos por segundo	p50	con error, de 360
CACHE_ON_ERROR=fail	40 de 40 con 500	1.5 ms	143 de 360 (39.7 %)
CACHE_ON_ERROR=degrade	40 de 40 con 200 (bypass)	8.3 a 10.0 ms	0 de 360 (0.0 %)

Degradar: el código y su costo

src/cache.ts

```ts
try {
  cached = await client.get(fullKey);
} catch (error) {
  if (variant.onCacheError === 'degrade') {
    counters.bypasses += 1;
    return { body: await compute(),
             outcome: 'bypass' };
  }
  counters.failures += 1;
  throw new CacheUnavailableError(error);
}
```

Trade-off

Cuando la cache se cae, la base de datos recibe de golpe toda la carga que la cache le ahorraba.

Si el origen solo aguanta con la cache puesta, degradar le traslada la caída. Fallar rápido puede ser la decisión correcta: se decide midiendo el servicio **sin** cache.

El cliente que espera en silencio

```
make offline-queue

Por omisión (con cola):
  con Redis caído, el GET sigue pendiente después de 2000 ms
  con Redis de vuelta, ese mismo GET respondió null a los 2219 ms
  un GET nuevo, ya reconectado, respondió null
Con disableOfflineQueue:
  con Redis caído, el GET falló a los 0 ms: The client is offline
  un GET nuevo, ya reconectado, respondió null
```

- Por omisión, el cliente de Redis guarda los comandos hasta que Redis vuelva: ni error ni respuesta.
- El `catch` que degrada nunca se ejecuta. **Sin detección no hay recuperación.**
- Todo cliente tiene una política para cuando el servicio no está. Hay que saber cuál es.

El retry: táctica y riesgo

Recupera de una falta transitoria con un bucle. Dos riesgos:

- **Duplicar la operación**: si se perdió la respuesta, el segundo intento cobra otra vez. *(Idempotencia: semana 12)*
- **Multiplicar la carga** sobre un servicio que ya no da más. *(Semana 14)*

Capas que reintentan

Con d capas y hasta a intentos en cada una:

$$a^d \text{ pedidos en la última capa}$$

Tres intentos en tres capas: $3^3 = 27$ llamadas a la pasarela por un clic.

El libro de SRE de Google (cap. 22) describe cómo los reintentos de lo rechazado se suman a la carga del segundo siguiente. Recomienda limitar los intentos y reintentar en una sola capa.

sre.google/sre-book/addressing-cascading-failures

SECCIÓN 07

Seguridad

07

Seguridad: tres propiedades y un escenario

- **Confidencialidad**: los datos solo los ve quien está autorizado.
- **Integridad**: solo los cambia quien está autorizado, y de las formas permitidas.
- **Disponibilidad**: el sistema responde a sus usuarios aunque alguien intente impedirlo.

El escenario

Fuente: un atacante, de fuera o de dentro.
Estímulo: el ataque.
Medida: cuánto daño, cuánto tardó en detectarse, cuánto en recuperarse.

"El sistema debe ser seguro" no se puede comprobar ni refutar: sin medida no hay escenario.

Tácticas de seguridad

Detectar

Intrusos
Denegación de servicio
Integridad de los mensajes

Resistir

Identificar, autenticar y autorizar actores
Limitar el acceso
Limitar la exposición
Cifrar los datos
Separar entidades
Validar la entrada
Cambiar las credenciales por defecto

Reaccionar

Revocar el acceso
Restringir el inicio de sesión
Informar

Recuperarse

Auditoría
Las tácticas de disponibilidad

Una alarma, una cerradura, un vigilante que actúa y un seguro para después.

Validar la entrada, medido

errors/sql-concat.ts

```ts
`... WHERE b.city = '${city}' ORDER BY 1`       // MAL: concatena
`... WHERE b.city = $1 ORDER BY 1`, [city]      // BIEN: parámetro
```

make sql-concat

```
city = Cusco
  concatenando: 6 filas; la primera: {"name":"Agua","author":"José María Arguedas"}
  con parámetro: 6 filas   (una ciudad)
city = x' OR '1'='1
  concatenando: 31 filas; la primera: {"name":"Agua","author":"José María Arguedas"}
  con parámetro: 0 filas   (todo el catálogo)
city = x' UNION SELECT rolname, 'superusuario: ' || rolsuper FROM pg_roles WHERE rolcanlogin --
  concatenando: 1 filas; la primera: {"name":"postgres","author":"superusuario: true"}
  con parámetro: 0 filas   (otra tabla)
```

Saltzer y Schroeder (1975): seis principios vigentes

Economía de mecanismo	Tan simple y pequeño como se pueda: sus errores no se ven en el uso normal
Valores por defecto seguros	Se decide por permiso, no por exclusión
Mediación completa	Todo acceso a todo objeto se comprueba, cada vez
Diseño abierto	La seguridad no depende de que el diseño sea secreto
Separación de privilegios	Dos llaves son más robustas que una
Mínimo privilegio	Cada programa opera con lo mínimo que necesita

Fallar haciendo ruido

Un error en un mecanismo que *da* permisos niega el acceso, y alguien se queja pronto. Un error en uno que *quita* permisos lo permite, y puede pasar inadvertido años.

Mínimo privilegio, medido

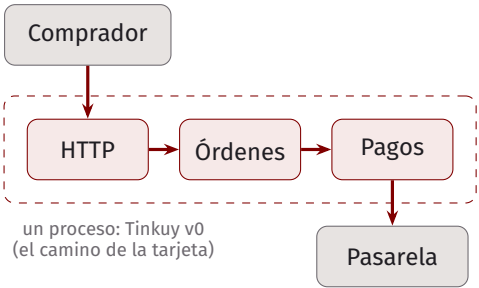
make privilege (fragmento)

```
SET ROLE w06_reports;
SET
SELECT count(*) AS prestados FROM copies WHERE status = 'loaned';
 prestados
-----------
      2905
(1 row)

UPDATE copies SET status = 'available';
psql:security/least-privilege.sql:24: ERROR:  permission denied for table copies
DROP TABLE copies;
psql:security/least-privilege.sql:25: ERROR:  must be owner of table copies
SELECT rolname, rolpassword FROM pg_authid LIMIT 1;
psql:security/least-privilege.sql:26: ERROR:  permission denied for table pg_authid
```

Lo rechaza la **base de datos**: no depende de que el código del servicio de reportes esté bien escrito. Táctica: limitar el acceso. Costo: un rol por servicio que crear y mantener.

La seguridad es estructura



Límite de confianza: la línea entre lo que el sistema controla y lo que no.

- ¿Qué componentes ven los datos más sensibles?
- ¿Quién habla con quién, y con qué credenciales?

Sacar pagos a un servicio *(P4)*: **separar entidades**.

Error común

Tratar la seguridad como una capa que se agrega al final. Un límite de confianza *es* estructura.

SECCIÓN 08

Una táctica cuesta otro atributo

08

No hay tácticas gratis

Táctica	Mejora	Cuesta
Diferir el enlace	Modificabilidad	Más combinaciones que probar
Pool de conexiones	Rendimiento	Un parámetro que elegir; un pool que se agota
Cache con TTL	Rendimiento	Datos viejos; una dependencia más
Redundancia	Disponibilidad	Dinero y consistencia
Retry	Disponibilidad	Rendimiento bajo sobrecarga; duplicados
Degradar a la base de datos	Disponibilidad	Carga sobre el origen, cuando la cache falta
Autenticar y cifrar cada llamada	Seguridad	Latencia y fricción
Mínimo privilegio	Seguridad	Roles y credenciales que mantener

Idea clave

Si al elegir una táctica no encuentran qué cuesta, todavía no terminaron de analizarla.

Laboratorio: pool de conexiones y cache, medir antes y después

En parejas, sobre **su copia de Tinkuy**; `code/week-06/` es la referencia.

1. Línea de base: `make load-browse` tres veces; conexiones en `pg_stat_activity`.
2. ¿Dónde se va el tiempo? Por endpoint y con `EXPLAIN ANALYZE`. Escribir la hipótesis.
3. Un `pg.Pool`, con tamaños 2, 10 y 50: el p95 contra el tamaño.
4. `GET /events` con una sola consulta. ¿Responde lo mismo?
5. Cache en Redis con TTL de 5 s. Comprar y mirar la lista: el trade-off.
6. La tabla de `results-template.md`: ¿cuál dio más por menos?
7. `svc kill redis` con carga. Degradar y repetir.
8. `security-checklist.md`: tres tácticas ausentes, por el daño que evitarían.

Para la próxima clase

- Sin entregas esta semana. Lo que midieron hoy es el punto de partida de **P2 · La apertura de ventas**.
- Terminar la tabla del laboratorio, con la columna de lo que costó cada táctica.
- Semana 7, decidir, documentar y evaluar. Leer de *Software Architecture in Practice* el cap. 22 *Documenting an Architecture*, y Michael Nygard, *Documenting Architecture Decisions* (2011).

Pregunta

Dentro de un año, alguien abre el código y encuentra un pool de 10 conexiones. ¿Cómo sabe por qué son 10 y no 50, y qué se midió para decidirlo?

Lecturas de hoy: *Software Architecture in Practice*, cap. 8, 9, 4 y 11; Saltzer y Schroeder, *The Protection of Information in Computer Systems* (1975).