

Arquitectura de Software

Estilos distribuidos

Service-based y microservicios · Las falacias de la red · Event-driven · Cómo se elige un estilo

Gerson Garrido

Semana 5

UNIVERSIDAD NACIONAL DE INGENIERÍA

Agenda

Sesión A

1. Por qué distribuir
2. Lo que cambia al cruzar la red
3. Service-based
4. Microservicios
5. Cómo se comunican los servicios

Sesión B

6. Eventos y comandos
7. El caso: después de una compra
8. Broker y mediator
9. Cómo se elige un estilo

Lectura

A: Richards y Ford, *Fundamentals of Software Architecture* (2.ª ed.), cap. 14 y 18; Lewis y Fowler, *Microservices* (2014).
B: *Fundamentals of Software Architecture*, cap. 15 y 19; Fowler, *What do you mean by "Event-Driven"?* (2017).

SECCIÓN 01

Por qué distribuir

01

Lo que un monolito da sin que nadie lo pida

- **Llamadas locales**: rápidas, y que no fallan a medias.
- **Una transacción**: varias escrituras ocurren todas o ninguna.
- **Un `JOIN`**: cualquier dato se combina con cualquier otro.
- **Un despliegue** y un solo proceso que mirar.

Esta semana

Cada estilo distribuido renuncia a una parte de esta lista. La pregunta es a cambio de qué.

Cada razón para distribuir es un escenario

Razón	El escenario que hay detrás
Escalar una parte sola	En una apertura de ventas, las compras se multiplican y los reportes no
Desplegar una parte sola	Un cambio en reportes llega a producción sin volver a desplegar la compra
Aislar una falla	Si el envío de correos se cae, la compra sigue
Equipos que no se pisan	Dos equipos despliegan la misma semana sin coordinarse
La carga no cabe en una máquina	Ni la máquina más grande atiende el pico

Sin escenario, no hay razón

Distribuir no es un objetivo: es el precio de un escenario que el monolito no cumple.

SECCIÓN 02

Lo que cambia al cruzar la red

02

Una llamada local y una remota, medidas

network/local-vs-remote.ts

```ts
function priceWithFee(cents: number): number {
  return Math.round(cents * 1.035) + 30;
}
```

make remote

```
llamada local:  7 ns por llamada (promedio de 1000000 llamadas)
llamada remota: 65 µs por llamada (mediana de 2000; p99 635 µs)
la remota tarda unas 9538 veces más, sin salir de la máquina
```

- Cuatro órdenes de magnitud, en la red más favorable que existe: la misma máquina.
- La remota ya no tiene *un* tiempo: tiene una distribución (el p99 es casi 10 veces la mediana).

Medido en el contenedor del curso, en una laptop con Apple Silicon. Lo que se traslada es la comparación.

Tinkuy pide un cobro y no recibe respuesta

¿Qué pudo pasar?

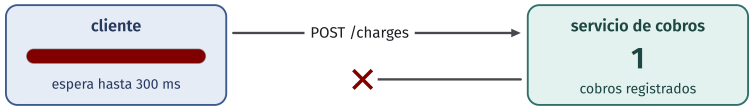
1. El pedido no llegó.
2. Llegó, y la pasarela murió antes de cobrar.
3. La pasarela cobró y murió antes de responder.
4. Cobró y respondió, y la respuesta se perdió.
5. Está viva, solo que más lenta de lo que Tinkuy espera. Va a cobrar.

Desde Tinkuy, los cinco se ven igual

En 1 y 2 no hubo cobro. En 3, 4 y 5 sí lo hubo, o lo va a haber.

Cuatro corridas de la misma llamada

1. none | 2. lost-request | 3. lost-reply | **4. slow**



EL CLIENTE VE

sin respuesta (TimeoutError)

EN EL SERVIDOR

se cobró

La respuesta sale a los 600 ms, cuando ya nadie la espera.

SALIDA DE MAKE LOST-REPLY

falla	el cliente ve	cobros en el servidor
none	respuesta 200	1
lost-request	sin respuesta (TypeError, UND_ERR_SOCKET)	0
lost-reply	sin respuesta (TypeError, UND_ERR_SOCKET)	1
slow	sin respuesta (TimeoutError)	1

Lo que ve quien llama, y lo que pasó

```
make lost-reply

none          el cliente ve: respuesta 200                                  cobros en el servidor: 1
lost-request  el cliente ve: sin respuesta (TypeError, UND_ERR_SOCKET)      cobros en el servidor: 0
lost-reply    el cliente ve: sin respuesta (TypeError, UND_ERR_SOCKET)      cobros en el servidor: 1
slow          el cliente ve: sin respuesta (TimeoutError)                   cobros en el servidor: 1
```

- Filas 2 y 3: el mismo error, y el servidor quedó en dos estados distintos.
- Fila 4: un timeout no cancela nada. El cliente se rindió; el servidor cobró igual.

Falla parcial

Una parte del sistema falló y la otra no tiene cómo saber cuál. Quien llama no distingue “no llegó” de “llegó y la respuesta se perdió”.

El dilema de quien no recibió respuesta

No reintentar

Puede quedar una compra sin cobrar.

Reintentar

Puede cobrar dos veces.

- Ninguna de las dos es segura por sí sola.
- La salida: que la operación se pueda repetir sin daño (**idempotencia**, semana 12).
- Cuánto esperar y cuándo dejar de insistir: timeouts, retries, circuit breaker (semana 14).

Las ocho falacias de la computación distribuida

1. La red es confiable
2. La latencia es cero
3. El ancho de banda es infinito
4. La red es segura
5. La topología no cambia
6. Hay un solo administrador
7. El costo de transporte es cero
8. La red es homogénea

Cómo leerlas

Cada una es algo que dentro de un monolito **es verdad**, y por eso nadie lo piensa. Al distribuir deja de serlo, y el diseño lo paga.

Peter Deutsch y otros, Sun Microsystems, años noventa. Fuente: *Fallacies of distributed computing* (Wikipedia).

Tres falacias en el Tinkuy de hoy

Falacia	El supuesto en el código
La red es confiable	El cliente de la pasarela hace un `fetch` sin fijar un timeout
La latencia es cero	`GET /events`: una consulta para la lista y una más por cada evento
El ancho de banda es infinito	`GET /events/1/seats`: los 2000 asientos en cada llamada

No son bugs

Las pruebas pasan. Son supuestos que dejan de cumplirse cuando la otra punta está lejos.

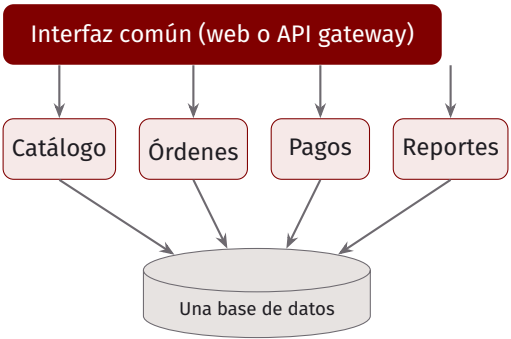
SECCIÓN 03

Service-based

03



Service-based: pocos servicios grandes



- Del orden de cuatro a doce servicios, uno por **dominio**.
- Cada uno se despliega por separado.
- Normalmente, la misma base de datos, a lo más partida por esquemas.
- El paso más corto desde un modular monolith.

Service-based: qué se gana y qué se paga

Se gana

- Despliegue por partes.
- Escalado por partes.
- Aislamiento parcial de fallas.
- Y se **conservan** las transacciones y los `JOIN`.

Se paga

- Algunas llamadas remotas.
- Varios despliegues que operar.
- La base de datos sigue siendo un punto único de falla y un contrato compartido.

El riesgo: si reportes lee con SQL propio las tablas de órdenes, cambiar una columna obliga a desplegar los dos. Defensa: cada tabla tiene un servicio que es su owner (un esquema por servicio).

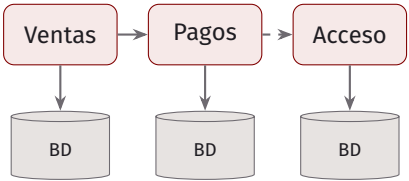
SECCIÓN 04

Microservicios

04



Microservicios: la forma



cada servicio, sus datos:
nadie más los lee

Lewis y Fowler (2014)

Una aplicación como un conjunto de servicios pequeños, cada uno en su propio proceso, que se comunican con mecanismos livianos, construidos alrededor de capacidades del negocio y desplegados de forma independiente.

Los autores aclaran que no hay una definición formal: es una lista de características que suelen ir juntas.

Cada servicio es owner de sus datos

La regla que define el estilo: nadie más lee la base de datos de un servicio.

- Mientras otro lea mis tablas, mi esquema es parte de mi interfaz.
- Sin datos independientes, no hay despliegue independiente.

El costo

- Ya no hay `JOIN` entre dominios.
- Ya no hay transacción entre servicios: no existe un `ROLLBACK` que deshaga los dos.

Idea clave

El límite de un servicio lo definen sus datos, no su código. Dos servicios que comparten tablas son un servicio con dos procesos.

Bounded context: una palabra, tres significados

Catálogo
asiento = lugar físico:
sección, fila, número

Órdenes
asiento = algo que se vende:
precio, disponible o vendido

Control de acceso
asiento = código de entrada
que se valida una vez

- Un bounded context es un modelo del dominio con significado propio, y un límite dentro del cual vale.
- Un solo modelo `Seat` para los tres termina con campos y reglas que se estorban.
- Los límites no son los sustantivos: no hay un “servicio de asientos”.

Lo que cuestan los microservicios

- **Los datos**: sin `JOIN` y sin transacciones entre servicios *(semanas 12 y 13)*.
- **Las fallas**: cada llamada puede fallar o tardar *(semana 14)*.
- **La operación**: un despliegue, una base de datos y unas alertas por servicio.
- **La observabilidad**: una compra cruza cinco procesos; hay que poder seguirla *(semana 14)*.
- **Los límites se vuelven caros de mover**: dos APIs, dos bases, dos despliegues.

> **Los propios autores**
>
> Lewis y Fowler cierran con “optimismo cauteloso” y recogen como razonable el consejo de empezar con un monolito, mantenerlo modular y partirlo cuando se vuelva un problema.

Granularidad: dos fuerzas

Empujan a partir

- Tasas de cambio distintas.
- Escalado distinto.
- Aislamiento: algo inestable, o datos con seguridad propia.

Empujan a unir

- Transacciones.
- Flujos que conversan mucho.
- Datos compartidos.

La granularidad correcta

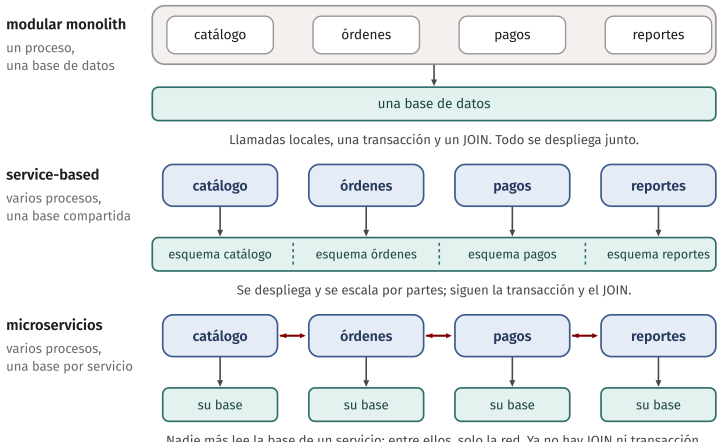
Deja las transacciones y las conversaciones frecuentes **dentro** de un servicio, y los cambios independientes **entre** servicios.

Service-based y microservicios, lado a lado

	Service-based	**Microservicios**
Servicios	Pocos y grandes	Muchos y pequeños
Un servicio es	Un dominio completo	Un solo propósito
Datos	Una base compartida	Una base por servicio
Transacciones	Las de la base de datos	Entre servicios no hay: sagas
Consultas cruzadas	Un `JOIN`, con disciplina	Por la red, o con copias
Operación	Moderada	Alta: exige automatización

Lo que se pierde al pasar a la derecha está en las filas de datos. Lo que se gana: ningún equipo depende del esquema de otro.

Los mismos dominios en tres estilos



SECCIÓN 05

Cómo se comunican los servicios

05

Síncrona y asíncrona

	Síncrona *(HTTP, gRPC)*	**Asíncrona** *(mensajes)*
Quien llama	Espera la respuesta	Deja el mensaje y sigue
Si el otro está lento	Quien llama está lento	Los mensajes se acumulan
Si el otro está caído	Quien llama falla	Los mensajes esperan
Resultado	Inmediato	No hay: el efecto llega después
Sirve cuando	Se necesita la respuesta para continuar	Solo se necesita que el trabajo se haga

Acoplamiento en el tiempo

Con una llamada síncrona, los dos servicios tienen que estar vivos y con capacidad en el mismo instante.

La disponibilidad de una cadena síncrona

Una compra necesita que respondan cuatro componentes, cada uno con 99,9 %:

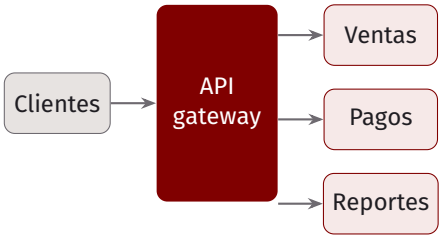
$$0{,}999^4 \approx 0{,}9960 \quad \text{es decir,} \quad 99{,}6\,\%$$

- Un componente: $0{,}001 \times 8760\ \mathrm{h} \approx 8{,}8$ horas caído al año.
- La cadena: $0{,}0040 \times 8760\ \mathrm{h} \approx 35$ horas al año.
- Cada servicio cumple su objetivo y el conjunto falla cuatro veces más.

En la industria

Lewis y Fowler (2014) cuentan que el sitio de *The Guardian* adoptó una regla sencilla: una sola llamada síncrona por petición de usuario. Lo que se traslada es el razonamiento, no la regla.

API gateway



Una entrada única que:

- **enruta** cada petición a su servicio;
- **autentica** una vez, en el borde;
- **limita** cuánto acepta de cada cliente.

Trade-off

Esconde la topología interna. A cambio: un salto más, un punto único de falla, y la tentación de meterle lógica del negocio. Un gateway no decide precios.

En Tinkuy: ¿qué parte saldría del monolito?

Pagos: el mejor candidato

- Dependen de un servicio externo que puede estar lento.
- Requisitos de seguridad propios.
- En contra: hoy cobrar y vender el asiento son una transacción.

Es el trabajo de P4.

Catálogo: parece, y no

- Se lee muchísimo más de lo que se escribe.
- Para “muchas lecturas” hay respuestas más baratas que un servicio: réplica y cache.

Es el trabajo de P3.

Y las **notificaciones**: el caso de la segunda sesión.

En Tinkuy: el correo va dentro de la compra

```
make tinkuy-email   y   EMAIL_LATENCY_MS=800 make tinkuy-email

Tinkuy inicial, EMAIL_LATENCY_MS=120 EMAIL_FAIL_EVERY=0
compra del asiento 1: 201 en 0.234284 s
compra del asiento 2: 201 en 0.216629 s
...
Tinkuy inicial, EMAIL_LATENCY_MS=800 EMAIL_FAIL_EVERY=0
compra del asiento 1: 201 en 0.913168 s
compra del asiento 2: 201 en 0.903385 s
...
```

- Cada milisegundo que tarda el correo lo espera el comprador.
- Se mide sobre una copia del Tinkuy inicial (`build/tinkuy`); se omiten tres compras de cada tanda.

En Tinkuy: y si el servidor de correo rechaza los mensajes

```
EMAIL_FAIL_EVERY=1 make tinkuy-email

Tinkuy inicial, EMAIL_LATENCY_MS=120 EMAIL_FAIL_EVERY=1
compra del asiento 1: 500 en 0.237319 s
compra del asiento 2: 500 en 0.216168 s
compra del asiento 3: 500 en 0.212930 s
compra del asiento 4: 500 en 0.213226 s
compra del asiento 5: 500 en 0.213978 s
órdenes nuevas en la base de datos: 0
cobros que la pasarela aprobó antes de esta sonda: 5 (la sonda recibió ch_00000006)
```

Cinco errores, ninguna orden, cinco cobros

El `ROLLBACK` deshizo las filas de Tinkuy. No pudo deshacer lo que ya pasó en otro sistema. Basta **un** sistema externo para tener fallas parciales.

Error común: el monolito distribuido

Servicios que conservan los acoplamientos del monolito. Tres señales; basta una:

1. **Comparten tablas.** Un cambio de esquema obliga a cambiar varios servicios.
2. **Hay que desplegarlos juntos.** ¿Se puede desplegar uno un martes cualquiera sin avisar?
3. **Se llaman en cadena de forma síncrona.** La latencia se suma y la disponibilidad se multiplica.

Los costos de los dos mundos

Llamadas remotas, fallas parciales y muchos despliegues, sin poder desplegar ni escalar una parte sola.

Error común: partir por sustantivos

Por sustantivos

“Servicio de usuarios”, “de órdenes”, “de asientos”, “de pagos”.

Una compra toca los cuatro, sin transacción.

Por capacidades del negocio

“Vender entradas”, “cobrar”, “controlar el acceso”.

Cada uno con los sustantivos que necesita, con su propio significado.

Un límite equivocado entre módulos se corrige moviendo archivos. Entre servicios: migrar datos, cambiar dos APIs y coordinar dos equipos. Por eso los límites se descubren **dentro** de un monolito.

SECCIÓN 06

Eventos y comandos

06



Evento y comando

	Comando	Evento
Dice	“Haz esto”	“Pasó esto”
Nombre	Imperativo: `ChargeCard`	Pasado: `OrderPaid`
Destinatario	Uno, conocido	Cero, uno o muchos, desconocidos
Quien envía espera	Que se haga, y un resultado	Nada
Se puede rechazar	Sí	No: ya pasó
Quién conoce a quién	Quien envía conoce al que recibe	Quien recibe conoce al que publica

La última fila

Con un evento, la dependencia se **invierte**: órdenes publica `OrderPaid` y no sabe nada del correo.

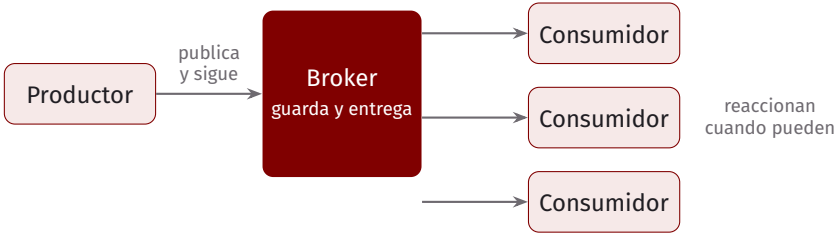
¿Evento o comando?

Mensaje	Es	Por qué
`SendConfirmationEmail`	comando	le encarga un trabajo a alguien
`SeatReserved`	evento	un hecho que ya ocurrió
`ReserveSeat`	comando	puede ser rechazado
`RefundRequested`	evento	el comprador *ya* lo pidió (el comando: `RefundPayment`)

Error común: el comando disfrazado

Publicar `OrderPaid` *para que* alguien envíe el correo es un comando con nombre de evento. Fowler lo llama comando pasivo-agresivo.

El estilo event-driven



La restricción del estilo

El productor no conoce a los consumidores. Lo único que comparten es el evento: su nombre y la forma de sus datos.

Event-driven: qué se gana y qué se paga

Se gana

- Desacoplamiento en el tiempo: el consumidor puede estar caído.
- Absorber picos: la cola espera, no el comprador.
- Agregar un consumidor no toca al productor.
- Un consumidor que falla no tumba a los demás.

Se paga

- Consistencia eventual.
- Flujos que no están escritos en ningún programa.
- Un manejo de errores que hay que diseñar.
- Un broker que operar, y del que todo depende.

La mejor manera de entender la lista es medirla en un caso.

SECCIÓN 07

El caso: después de una compra

07

El caso de `code/week-05/`

Un sistema mínimo de venta de entradas, separado de Tinkuy. Después de cada compra hay tres **efectos**:

1. enviar el **correo** de confirmación *(simula la demora con `EMAIL_LATENCY_MS`, 100 ms)*;
2. actualizar el **reporte** de ventas;
3. sumar **puntos** al comprador *(falla siempre con `POINTS_FAIL=1`)*.

`sync/`

Órdenes llama a los tres, uno tras otro.

`events/`

Órdenes publica `OrderPaid` en NATS y tres consumidores reaccionan.

Mismos efectos, mismo servidor HTTP, misma base de datos. Cambia cómo se entera cada efecto.

La versión síncrona

sync/place-order.ts

```ts
import { sendConfirmationEmail } from '../effects/email.ts';
import { addLoyaltyPoints } from '../effects/loyalty-points.ts';
import { addToSalesReport } from '../effects/sales-report.ts';
import { savePaidOrder } from '../shared/orders-store.ts';
import type { NewOrder, PaidOrder } from '../shared/types.ts';

export async function placeOrder(input: NewOrder): Promise<PaidOrder> {
  const order = await savePaidOrder(input);
  await sendConfirmationEmail(order);
  await addToSalesReport(order);
  await addLoyaltyPoints(order);
  return order;
}
```

Dice todo lo que pasa después de una compra, y en qué orden. Responde cuando termina la última línea.

La versión con eventos

events/place-order.ts

```ts
import { savePaidOrder } from '../shared/orders-store.ts';
import { publishOrderPaid } from './bus.ts';

export async function placeOrder(input: NewOrder): Promise<PaidOrder> {
  const order = await savePaidOrder(input);
  await publishOrderPaid({
    type: 'OrderPaid', id: randomUUID(), occurredAt: new Date().toISOString(), order,
  });
  return order;
}
```

Órdenes no importa ningún efecto ni sabe cuántos hay. Abreviado: se omiten dos `import` y el evento va en una línea.

Los consumidores y el contrato

events/consumers/email.ts

```ts
import { sendConfirmationEmail } from '../../effects/email.ts';
import { runConsumerProcess } from '../consumer-process.ts';

await runConsumerProcess('email', (event) => sendConfirmationEmail(event.order));
```

events/order-paid.ts

```ts
export type OrderPaid = {
  type: 'OrderPaid';
  id: string;
  occurredAt: string;
  order: PaidOrder;
};
```

Un proceso por efecto. Lo único que comparten con el productor es el tipo del evento: el contrato.

Quién conoce a quién



- Las flechas son `import`. En `events/`, las dependencias se invirtieron.
- Cuarto efecto: en `sync/` se modifica el archivo central de órdenes; en `events/`, ningún archivo existente.
- El acoplamiento no desapareció: cambió de lugar. Ahora lo caro de cambiar es `OrderPaid`.

NATS JetStream en una diapositiva



- **Stream**: guarda en disco cada mensaje publicado, haya o no alguien escuchando.
- **Consumer durable**: un cursor con nombre, guardado en el servidor. Uno por efecto.
- **Ack**: confirma que el mensaje se procesó. Sin ack, el servidor lo vuelve a entregar.
- En el ejemplo: primero el trabajo, después el ack.

Cuánto tarda la compra

sync: make measure N=10

```
compra  1: 201 en   144 ms
compra  2: 201 en   107 ms
...
compra 10: 201 en   111 ms
10 compras, 0 con error. Tiempo de respuesta:
  mín 107 ms · p50 108 ms ·
  p95 144 ms · máx 144 ms
Los tres efectos de todas las órdenes estaban
  hechos 0.0 s después de la última respuesta.
```

events: make measure N=10

```
compra  1: 201 en    34 ms
compra  2: 201 en     4 ms
...
compra 10: 201 en     2 ms
10 compras, 0 con error. Tiempo de respuesta:
  mín 2 ms · p50 2 ms ·
  p95 34 ms · máx 34 ms
Los tres efectos de todas las órdenes estaban
  hechos 1.0 s después de la última respuesta.
```

- La mediana pasó de 108 ms a 2 ms. El trabajo no desapareció: diez correos de 100 ms son un segundo.
- Cambió **quién espera**: antes, el comprador; ahora, el evento en el stream.

Salidas abreviadas, con las líneas largas partidas. Con diez muestras, el p95 es el máximo.

Un paso lento: `EMAIL_LATENCY_MS=800`

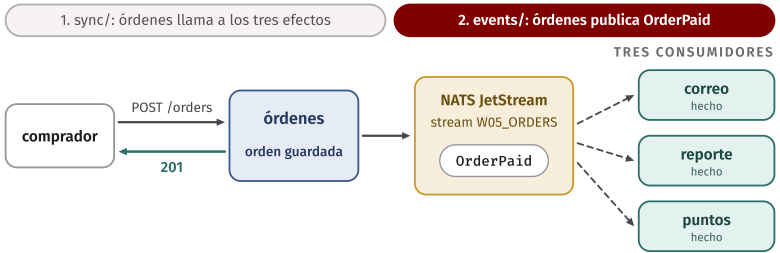
Mediana de la compra	Correo de 100 ms	Correo de 800 ms
`sync/`	108 ms	810 ms
`events/`	2 ms	2 ms
Efectos completos, en `events/`	1,0 s después	8,1 s después

```
make status, mientras el consumidor de correo trabaja

órdenes pagadas          10
correos enviados          3 de 10     correos (faltan 7)
reporte de ventas        10 de 10     órdenes (completo)
puntos sumados         1590 de 1590   puntos (completo)
```

Eso es **consistencia eventual** con números: las órdenes dicen diez, el correo dice tres, y los dos tienen razón.

La misma compra en las dos versiones



El correo tardó los mismos 100 ms. Ya no los espera el comprador: los espera el evento.

el comprador espera **2 ms** el correo: lo espera el evento, no el comprador

MEDIANA DE LA COMPRA, MAKE MEASURE N=10

	correo de 100 ms	correo de 800 ms	al responder, los efectos
`sync/`	`108 ms`	`810 ms`	ya ocurrieron
`events/`	`2 ms`	`2 ms`	todavía no: llegan después

El mismo instante, visto desde el broker

make info C=email (sección State)

```
Last Delivered Message: Consumer sequence: 4 Stream sequence: 4 Last delivery: 53ms ago
  Acknowledgment Floor: Consumer sequence: 3 Stream sequence: 3 Last Ack: 54ms ago
      Outstanding Acks: 1 out of maximum 1,000
  Redelivered Messages: 0
  Unprocessed Messages: 6
         Waiting Pulls: 0 of maximum 512
```

- `Acknowledgment Floor` 3: tres mensajes confirmados.
- `Outstanding Acks` 1: uno entregado y sin confirmar (se está procesando).
- `Unprocessed Messages` 6: seis que este consumer todavía no recibió. $3 + 1 + 6 = 10$.
- Si `Unprocessed Messages` crece sin parar, el consumidor no da abasto: la cola absorbe un pico, no un consumidor más lento que la llegada promedio.

Un paso que falla, en `sync/`

```
POINTS_FAIL=1 make sync   make buy N=3   make status

compra  1: 500 en   144 ms  {"error":{"code":"INTERNAL_ERROR","message":"Ocurrió un error inesperado"}}
compra  2: 500 en   108 ms  {"error":{"code":"INTERNAL_ERROR","message":"Ocurrió un error inesperado"}}
compra  3: 500 en   111 ms  {"error":{"code":"INTERNAL_ERROR","message":"Ocurrió un error inesperado"}}
3 compras, 3 con error. Tiempo de respuesta: mín 108 ms · p50 111 ms · p95 144 ms · máx 144 ms
órdenes pagadas          3
correos enviados         3 de 3      correos (completo)
reporte de ventas        3 de 3      órdenes (completo)
puntos sumados           0 de 385    puntos (faltan 385)
```

- El comprador ve un 500. La orden está pagada, el correo salió, el reporte la cuenta.
- Si reintenta, queda con dos órdenes pagadas.
- Los puntos de la primera no llegan nunca: nadie quedó encargado de reintentarlos.

Un paso que falla, en `events/`

Las tres compras responden 201. El mensaje que no se pudo procesar **sigue ahí**:

log de make events (líneas del consumidor de puntos)

```
[points] orden 1: intento 1 falló (Error: El programa de puntos no responde)
[points] orden 1: intento 2 falló (Error: El programa de puntos no responde)
[points] orden 1: intento 3 falló (Error: El programa de puntos no responde)
```

Consumer	Ack Pending	Redelivered	Unprocessed	Ack Floor
email	0	0	0	3
points	3	3	0	0
report	0	0	0	3

Columnas de `nats consumer report W05_ORDERS`, unos segundos después de las compras.

Al reiniciar el consumidor sin la falla, los puntos llegan (385 de 385). El stream hizo de **memoria del trabajo pendiente**.

Un consumidor detenido

Diez compras procesadas, `make stop C=email`, cinco compras más:

```
make status   y luego   make consumer C=email

correos enviados        10 de 15     correos (faltan 5)
...
[email] esperando eventos OrderPaid
[email] orden 11: correo enviado a rosa.quispe@correo.pe
[email] orden 12: correo enviado a luis.mamani@correo.pe
[email] orden 13: correo enviado a carmen.huaman@correo.pe
[email] orden 14: correo enviado a rosa.quispe@correo.pe
[email] orden 15: correo enviado a luis.mamani@correo.pe
```

No se perdió ninguno ni se repitió ninguno. Por dos decisiones de diseño:

1. el stream guarda los mensajes en disco;
2. el cursor vive en el servidor, no en el proceso.

Es una propiedad de este diseño, no de “los eventos”: sin persistencia, los cinco se pierden.

Lo que no resuelve (1): dos operaciones

Guardar la orden y publicar el evento no van en una transacción. Con dos compras hechas, `svc kill nats` y dos más:

```
make buy N=2   make status

compra  1: 500 en  5033 ms  {"error":{"code":"INTERNAL_ERROR","message":"Ocurrió un error inesperado"}}
compra  2: 500 en  5011 ms  {"error":{"code":"INTERNAL_ERROR","message":"Ocurrió un error inesperado"}}
2 compras, 2 con error. Tiempo de respuesta: mín 5011 ms · p50 5011 ms · p95 5033 ms · máx 5033 ms
órdenes pagadas          4
correos enviados         2 de 4      correos (faltan 2)
reporte de ventas        2 de 4      órdenes (faltan 2)
puntos sumados         340 de 680    puntos (faltan 340)
```

- Las órdenes 3 y 4 quedaron pagadas y sin evento. Al volver NATS, siguen incompletas.
- El broker, que iba a aislar a la compra, es ahora una dependencia síncrona de la compra.
- La solución (tabla outbox) es tema de la semana 12.

Lo que no resuelve (2, 3 y 4)

- **Un mensaje puede llegar dos veces.** Si el consumidor muere entre el trabajo y el ack, el efecto se repite. Los consumidores de esta semana son ingenuos a propósito: una orden de S/ 320 puede dejar 640 puntos *(hay una prueba que lo demuestra)*.
- **Pueden llegar en otro orden.** Un mensaje que falla se reintenta después de los que venían detrás.
- **Nadie sabe si los tres efectos ya ocurrieron.** Un 201 significa "la orden está pagada y el evento está en el stream". `make status` lo tuvimos que escribir.

> **Las tres preguntas de todo diseño con eventos**
>
> ¿Qué pasa si el consumidor falla a la mitad? ¿Y si el mensaje llega dos veces? ¿Y si llegan en otro orden? *(semanas 12 y 13)*

SECCIÓN 08

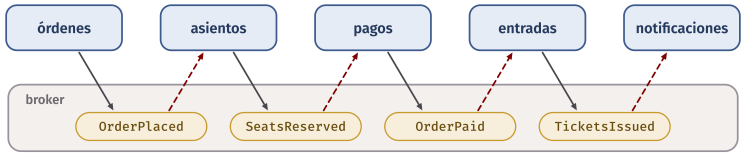
Broker y mediator

08



El mismo proceso en las dos topologías

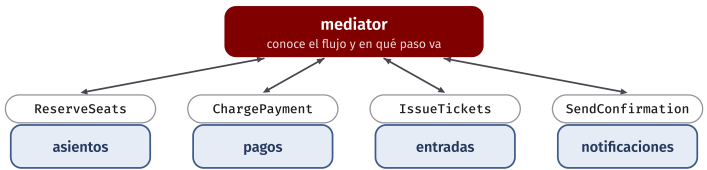
BROKER: CADA UNO ESCUCHA UN EVENTO, HACE SU PARTE Y PUBLICA OTRO



Flecha continua: publica. Flecha punteada: lo recibe porque está suscrito.

Ningún componente conoce a otro, y el flujo completo no está escrito en ningún lugar.

MEDIATOR: UN COORDINADOR ENVÍA COMANDOS Y ESPERA CADA RESPUESTA



y ReleaseSeats si el cobro falló

El flujo está escrito en un lugar; todos dependen del coordinador.

Topología broker: una cadena de reacciones

Cada componente escucha un evento, hace su parte y publica otro: `OrderPlaced`, `SeatsReserved`, `OrderPaid`, `TicketsIssued`.

Se gana

Nadie conoce a nadie. Agregar una reacción no toca a ninguno.

Se paga: nadie tiene el flujo

- ¿En qué estado está la compra 4812?
- Falló el cobro: ¿quién libera los asientos?

Topología mediator: un coordinador que conoce el flujo

1. `ReserveSeats` → asientos
2. `ChargePayment` → pagos
3. Si falló: `ReleaseSeats`
4. `IssueTickets` → entradas
5. `SendConfirmation`

El mediator envía **comandos**, no eventos, y espera cada respuesta.

Trade-off

Se gana control: el flujo está en un lugar y se sabe en qué va cada proceso. Se paga acoplamiento con el coordinador, que puede ser cuello de botella.

Broker o mediator

	Broker	Mediator
Quién conoce el flujo	Nadie	El mediator
Mensajes	Eventos	Comandos
Agregar una reacción	No toca a nadie	Cambia el mediator
Estado de un proceso	Difícil de saber	El mediator lo sabe
Errores y compensación	Repartidos	Centralizados

Regla práctica

Reacciones independientes a un hecho (notificar, reportar, sumar puntos): broker. Un proceso que debe terminar completo o deshacerse: mediator. En las sagas (semana 12): coreografía y orquestación.

“Usamos eventos” puede querer decir cuatro cosas

Fowler, *What do you mean by “Event-Driven”?* (2017):

1. **Notificación de eventos**: avisar que algo cambió; el evento lleva poco.
2. **Transferencia de estado en el evento**: el evento lleva los datos; el receptor no vuelve a preguntar. Es el `OrderPaid` de la semana.
3. **Event sourcing**: el log de eventos es la fuente de verdad; el estado se deriva.
4. **CQRS**: estructuras separadas para leer y para escribir.

Antes de discutir si conviene, hay que decir cuál

Los dos últimos se ven en la semana 13. Publicar `OrderPaid` no es event sourcing.

SECCIÓN 09

Cómo se elige un estilo

09

No hay estilo mejor: hay un procedimiento

1. Tomar los escenarios **(A, A)** del utility tree.
2. Para cada estilo candidato, decir **cómo** cumple cada escenario, o por qué no.
3. Sumar el **costo**: operación, tamaño del equipo, experiencia.
4. Elegir, y **escribir qué se sacrifica**.

Dos precisiones

Un sistema puede usar un estilo distinto en cada parte. Y los candidatos se comparan siempre contra **no cambiar nada**.

Un ejemplo: los efectos después de la compra

Escenario	`sync/`	`events/`
E1: correo en 800 ms, compra en menos de 300 ms	No: mediana 810 ms	Sí: mediana 2 ms
E2: puntos caído, la compra se completa	No: 500, y los puntos se pierden	Sí: 201, y los puntos llegan después
E3: efecto nuevo sin tocar órdenes	No	Sí
Costo	Ninguno nuevo	Broker, procesos, consistencia eventual, duplicados, saber qué ocurrió

Si E1, E2 y E3 son hipotéticos, la columna de la izquierda, con su costo nulo, gana.

La pregunta que más descarta

¿El equipo puede operar esto?

- Veinte servicios son veinte despliegues, veinte tableros y veinte cosas que despiertan a alguien.
- Cinco personas con veinte servicios: cuatro por persona, y nadie conoce bien ninguno.
- La arquitectura tiene que caber en el equipo que hay *(ley de Conway, semana 15)*.

Error común

Eventos donde quien llama necesita la respuesta para continuar: una llamada síncrona disfrazada.

Error común

Elegir el estilo por la empresa que lo hizo famoso. Su contexto es parte de la decisión.

Laboratorio: el mismo caso en dos estilos

En parejas, con `code/week-05/` y dos terminales en el contenedor.

1. Leer las dos versiones y dibujar quién conoce a quién.
2. `make sync`, `make events`; `make measure N=10` en cada una.
3. `EMAIL_LATENCY_MS=800`: ¿a dónde va la latencia?
4. `POINTS_FAIL=1`: ¿qué recibe el comprador?, ¿qué muestra `make info C=points`?
5. Detener un consumidor, cinco compras, iniciarlo de nuevo.
6. Agregar un cuarto efecto (auditoría) y contar los archivos existentes que tocaron.
7. Tres escenarios y una tabla: qué cumple cada versión y a qué costo.
8. Discusión: para Tinkuy, hoy, ¿cuál?

Para la próxima clase

- Esta semana se entrega **P1 · Módulos con límites** y se publica **P2 · La apertura de ventas**.
- Terminar la tabla del laboratorio, con sus tres escenarios.
- Semana 6, tácticas. Leer de *Software Architecture in Practice*: cap. 8 *Modifiability* y cap. 9 *Performance* (Sesión A); cap. 4 *Availability* y cap. 11 *Security* (Sesión B).
- Lectura libre: Saltzer y Schroeder, *The Protection of Information in Computer Systems* (1975), la sección de principios de diseño.

Pregunta

En la versión con eventos, ¿cómo sabe alguien que el correo de la orden 7 ya salió? ¿Quién debería poder responderlo?

Lecturas de hoy: *Fundamentals of Software Architecture* (2.ª ed.), cap. 14, 15, 18 y 19; Lewis y Fowler, *Microservices*; Fowler, *What do you mean by "Event-Driven"?*