

SISTEMAS DISTRIBUIDOS

CLASE 13 — SISTEMAS DE MENSAJERÍA

Contenido

1. Comunicación indirecta
2. Desacople y modelos de integración
3. Mensajería directa, message brokers y SQS
4. Colas de trabajo y dead-letter queues
5. Publisher-subscriber y SNS
6. Implementación de un broker de colas
7. Event streaming y logs particionados

COMUNICACIÓN INDIRECTA

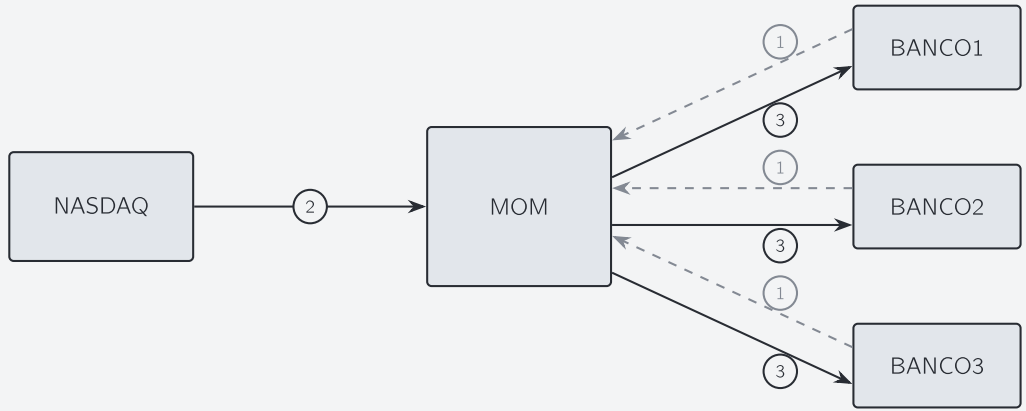
Comunicación directa



- El banco consulta al NASDAQ por **RPC**, periódicamente: **polling**
- Consultar seguido carga al servidor; consultar poco pierde actualizaciones
- 100 bancos × 5.700 símbolos/s: **570.000** requests/s para ~**3.000** novedades

Comunicación indirecta

- Invertir la flecha: el NASDAQ **avisa**
- En el medio, un intermediario: el **MOM**
- Reparte a todos los interesados: **publisher-subscriber**



1 CADA BANCO: `subscribe("stocks.NASDAQ.*")`

2 EL NASDAQ, CUANDO CAMBIA UN PRECIO: `push("stocks.NASDAQ.GOOGL", 100)`

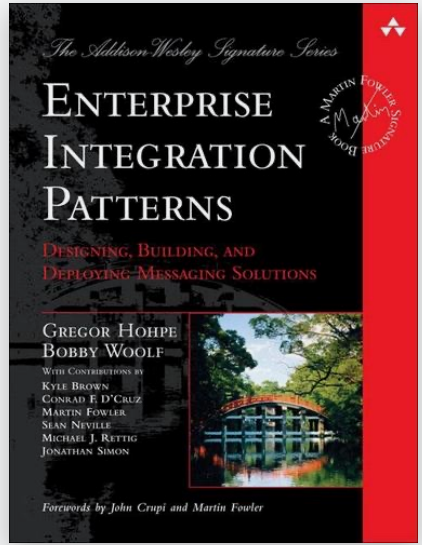
3 EL MOM SE LO ENTREGA A CADA BANCO SUSCRIPTO

Historia de la terminología

- **80**: los sistemas distribuidos se construyen con **RPC**
- **90**: aparecen los **MOM**; *middleware* se vuelve término comercial
- Ejemplo de la época: **TIBCO**, estándar de la mensajería financiera
- **2000**: *application servers* y **SOA**; **2010**: **microservicios**
- La idea del intermediario sigue vigente a pesar de la terminología

Mensajería como integración empresarial

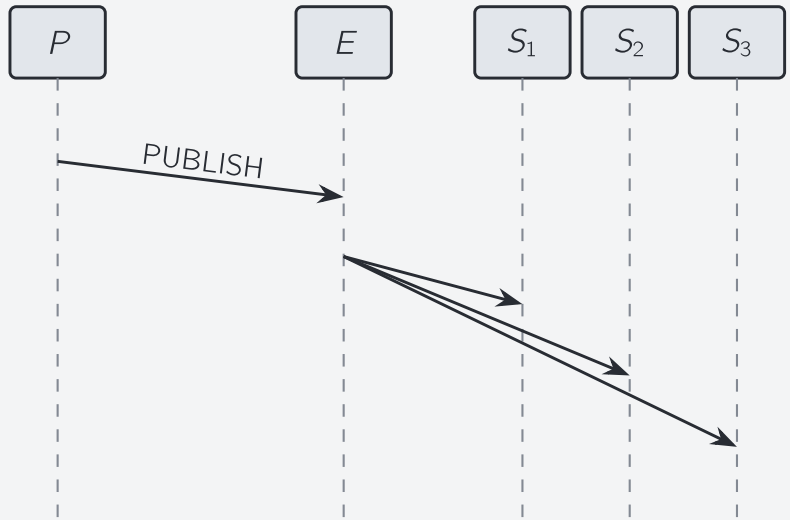
- Los primeros MOM: sistemas de **comunicación**, como el market data de TIBCO
- Hoy: sistemas de **integración** entre aplicaciones y equipos que no se conocen
- Los modelos de la clase son **patrones de integración**
- Lectura sugerida: *Enterprise Integration Patterns*, Hohpe y Woolf, Addison-Wesley, 2003



DESACOPLE Y MODELOS DE INTEGRACIÓN

Desacople en espacio

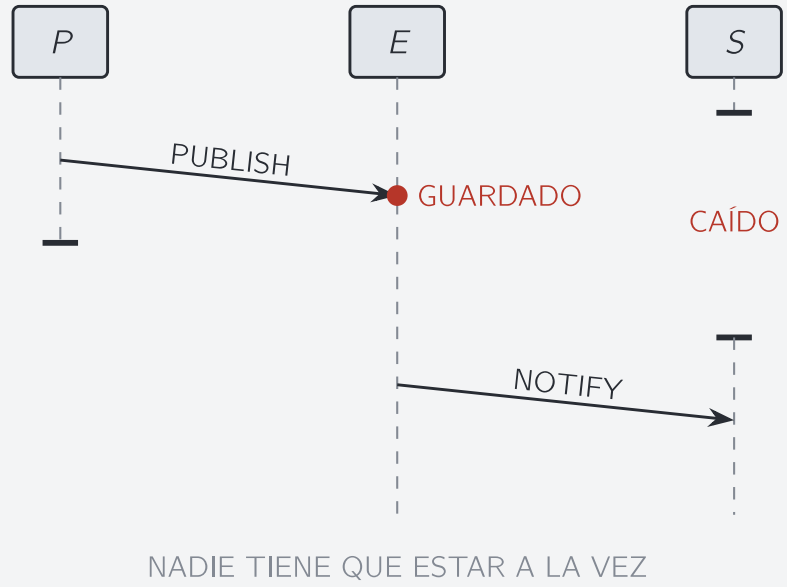
- El publisher no conoce la identidad ni la ubicación de los subscribers
- Publica en un **tópico**; el servicio de eventos conoce a los destinatarios
- Tampoco el equipo del publisher necesita conocer a los equipos que consumen



Clasificación de Eugster, Felber, Guerraoui y Kermarrec, «The Many Faces of Publish/Subscribe», *ACM Computing Surveys* 35(2), 2003

Desacople en tiempo

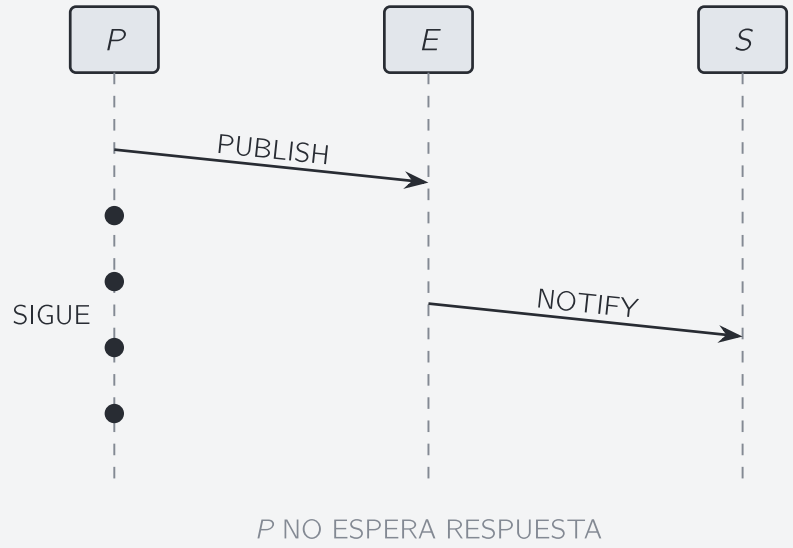
- Emisor y receptor no tienen que estar activos **a la vez**
- El servicio **guarda** el mensaje hasta que el receptor vuelve
- Contrapartida: una falla también se informa **desacoplada en el tiempo**



Clasificación de Eugster, Felber, Guerraoui y Kermarrec, «The Many Faces of Publish/Subscribe», *ACM Computing Surveys* 35(2), 2003

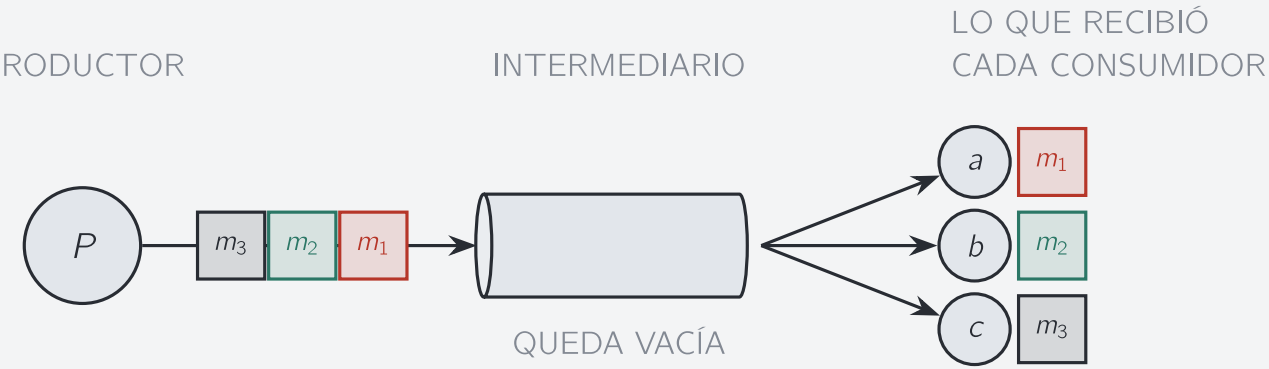
Desacople en sincronización

- El emisor **no espera** la respuesta: *send and forget*
- Un RPC suele ser sincrónico: se manda el pedido y se espera el resultado



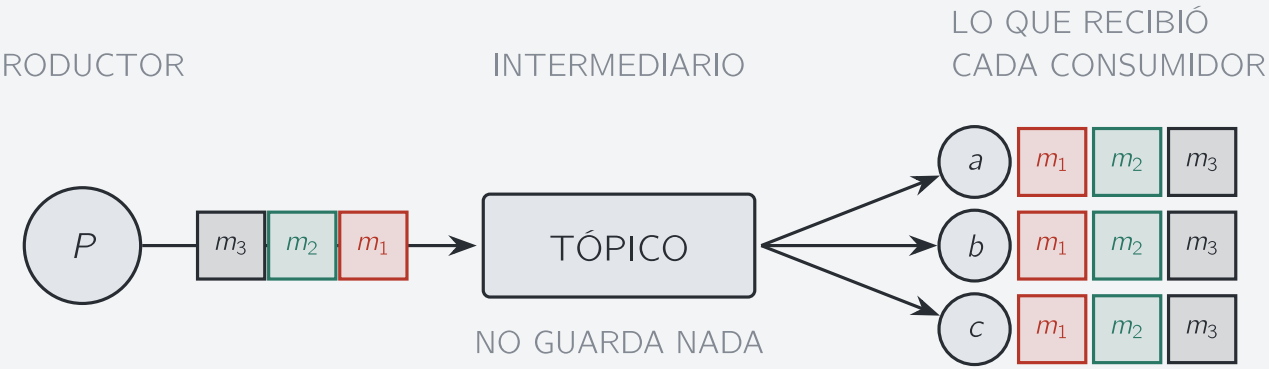
Clasificación de Eugster, Felber, Guerraoui y Kermarrec, «The Many Faces of Publish/Subscribe», *ACM Computing Surveys* 35(2), 2003

Modelo producer-consumer



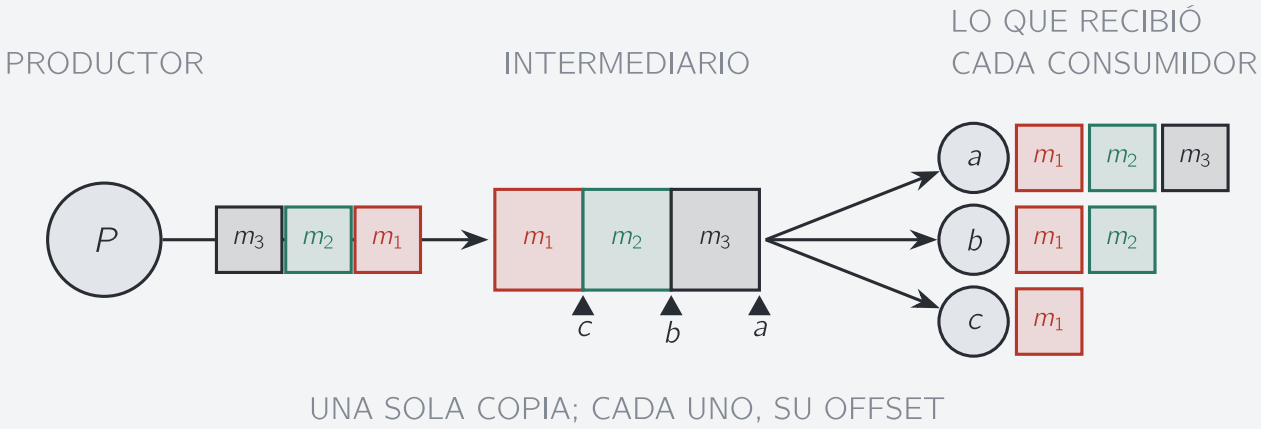
- Cada mensaje le llega a **un solo** consumidor, y se borra
- Ejemplo: **SQS**

Modelo publisher-subscriber



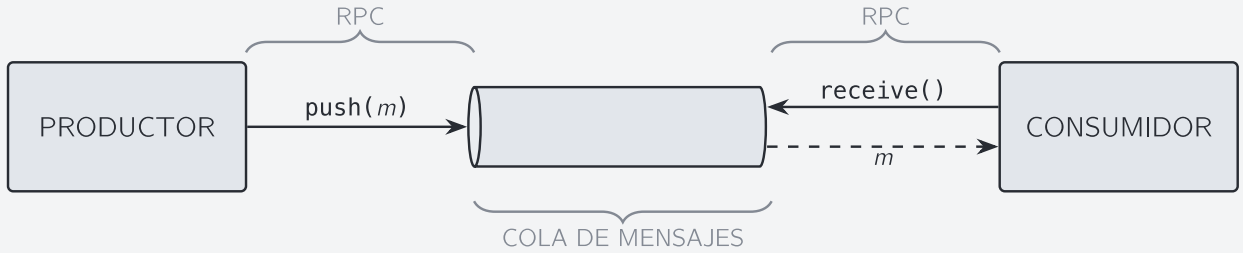
- Cada mensaje se **copia a todos** los suscriptores, en el momento, y no queda
- Ejemplo: **SNS**

Modelo event streaming



- Los mensajes **quedan en el log**: una sola copia
- Cada consumidor lee a su ritmo, desde su **offset**, aunque llegue después
- Ejemplo: **Kafka**

Producer-consumer: las flechas son RPC

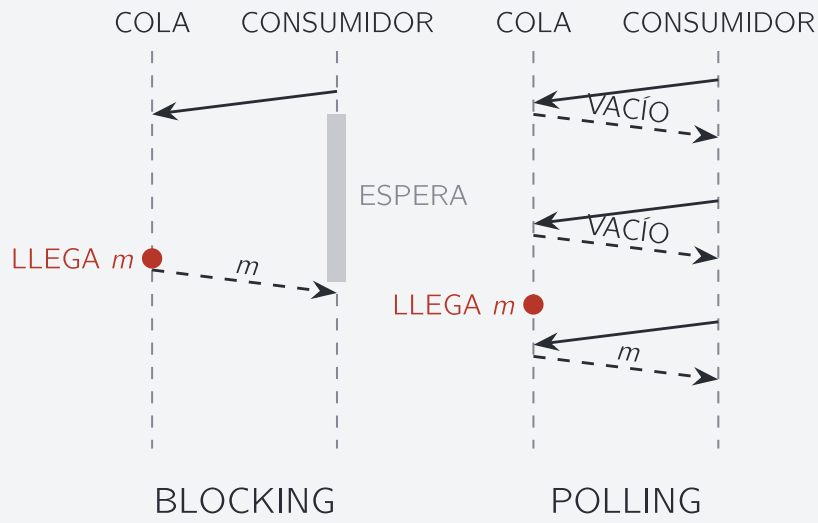


- También llamado **point-to-point**: cada mensaje va a un solo receptor
- `push` y `receive` son **RPC**: la cola es un sistema distribuido
- La cola no es necesariamente **FIFO**

Point-to-Point Channel: Hohpe y Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2003

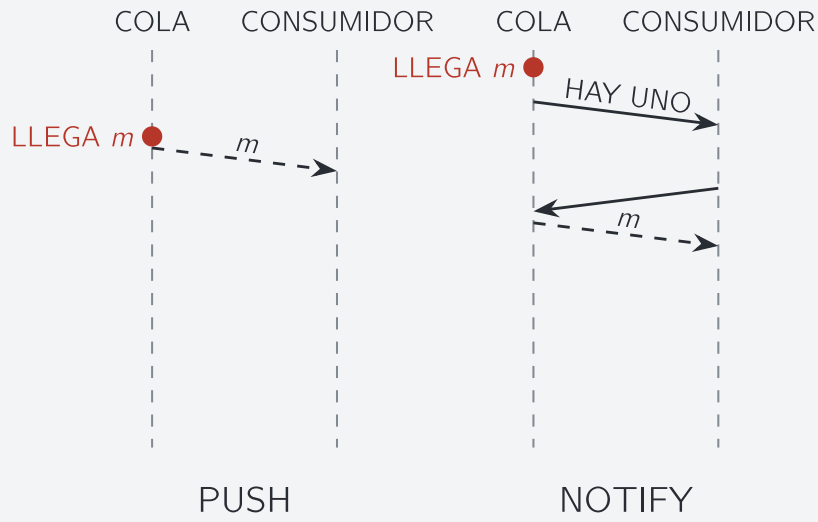
Recepción por pull: blocking y polling

- El **consumidor** toma la iniciativa: pide
- **Blocking**: espera hasta que aparezca un mensaje
- **Polling**: si no hay nada, vuelve a preguntar más tarde
- SQS clásico hace polling; con **long polling**, bloquea



Recepción por push: push y notify

- La **cola** toma la iniciativa
- **Push**: le manda el mensaje al consumidor
- **Notify**: avisa que hay uno, y el consumidor lo viene a buscar
- Ejemplo de push: **SNS** (bloque 5)



BIBLIOTECAS, BROKERS Y SQS

Mensajería directa

- La cola no existe en ninguna parte: la emula una **biblioteca**
- El productor se comunica directamente con cada consumidor
- El programador ve una cola; la red no
- Ejemplo: **ZeroMQ** (ØMQ), construido sobre la API de **sockets**

Message brokers

- Un **sistema dedicado** hace de intermediario
- Nube: **SQS** y **SNS** (AWS), **Pub/Sub** (Google), **Service Bus** (Azure)
- Instalables: **RabbitMQ**, **ActiveMQ**
- Son esencialmente un tipo de **base de datos**, no relacional
- Borran los mensajes al entregarlos: son **efímeros**
- Sin índices sofisticados, y pensados para **poco volumen** de storage

API de SQS

```
sqs.send_message(QueueUrl=url, MessageBody="tarea 1")
# devuelve: MessageId, MD5OfMessageBody

sqs.receive_message(QueueUrl=url)
# cada mensaje trae: MessageId, ReceiptHandle, Body

sqs.delete_message(QueueUrl=url, ReceiptHandle=recibo)
# recibo = ReceiptHandle devuelto por receive_message
```

Visibility timeout

- Al entregar un mensaje, la cola **no lo borra**: le arranca un timer, p. ej. **60 s**
- Mientras corre, el mensaje es **invisible** para los demás consumidores
- Si el consumidor muere, el timer expira y el mensaje **reaparece**
- Uso correcto: **recibir, procesar, borrar**
- Los consumidores compiten: efecto de **balanceo de carga**

Consumidor lento y duplicados

- Si procesar tarda más que el timeout, otro consumidor toma el mismo mensaje
- `ChangeMessageVisibility` extiende el timeout: un **heartbeat** sobre el mensaje
- Aun así hay duplicados: entrega **at-least-once**
- El consumidor tiene que ser **idempotente**

Bucle del consumidor

```python
# consumidor.py
while True:
    resp = sqs.receive_message(QueueUrl=url)
    mensajes = resp.get("Messages", [])
    if not mensajes:
        continue
    msg = mensajes[0]
    procesar(msg)          # tu lógica
    sqs.delete_message(QueueUrl=url, ReceiptHandle=msg["ReceiptHandle"])
```

EJEMPLO DE APLICACIÓN: COLAS DE TRABAJO

Pipeline de transcodificación de video

- El usuario sube un video y queda en un **storage de blobs**: S3 (o HDFS)
- Encola un mensaje pequeño en SQS: `transcodificar(path_s3)`
- Una flota de **workers** con FFmpeg compite por los mensajes
- Cada worker guarda en S3 la versión transcodificada

Poison pill y dead-letter queue

- Un mensaje que hace fallar a **cualquier** worker que lo tome
- El visibility timeout lo reentrega indefinidamente
- La cola cuenta las entregas de cada mensaje (`ApproximateReceiveCount`)
- Al superar el máximo (`maxReceiveCount`) lo mueve a una **dead-letter queue**
- Sin ese límite: 4 días de retención ÷ 60 s = **5.760 entregas**

PUBLISHER-SUBSCRIBER Y SNS

SNS y fan-out

- El mismo mensaje se entrega a **todos** los suscriptores: **fan-out**
- Para consumidores **heterogéneos**: otros servicios, equipos o compañías
- SNS hace **push**: invoca el endpoint del suscriptor
- Receptor no disponible: **retry N veces**, y después descarte o dead-letter queue
- Dos casos de uso: **app-to-app** y **app-to-person** (email, SMS, push)

API de SNS

```
sns.subscribe(TopicArn=arn, Protocol="sqs", Endpoint=cola_arn)
# Protocol: sqs, lambda, http, email, sms...

sns.publish(TopicArn=arn, Message="hola")
# fan-out a todos los suscriptores
```

- No hay `receive`: SNS empuja
- Un suscriptor recibe solo lo publicado **después** de suscribirse

UN BROKER DE COLAS DESDE CERO

Opción 1: Postgres como cola

ANTES

id	mensaje	visibility_timeout	
1	m_1	09:59:30	VISIBLE
2	m_2	10:00:40	INVISIBLE
3	m_3	09:59:58	VISIBLE

VISIBLE = `visibility_timeout < now()`

10:00:00 ⟹

DESPUÉS DEL RECEIVE

id	mensaje	visibility_timeout	
1	m_1	10:01:00	INVISIBLE
2	m_2	10:00:40	INVISIBLE
3	m_3	09:59:58	VISIBLE

EL RECEIVE LO CORRE A `now() + 60 s`

- Un broker **stateless** atiende los RPCs; la cola es una tabla de Postgres
- Un mensaje es **visible** si su `visibility_timeout` ya pasó
- Extensiones como **pgmq** lo traen resuelto, al estilo de SQS

Opción 1: send y receive en SQL

```
-- send: encola un mensaje, visible de entrada
INSERT INTO cola (mensaje, visibility_timeout) VALUES ($1, now());

-- receive: toma un mensaje visible y lo esconde 60 segundos
UPDATE cola
   SET visibility_timeout = now() + interval '60 seconds'
 WHERE id = (SELECT id FROM cola
             WHERE visibility_timeout < now()
             ORDER BY id
               FOR UPDATE SKIP LOCKED
             LIMIT 1)
RETURNING id, mensaje;
```

Opción 2: sharding aleatorio

- Varios pares broker + base; send y receive eligen un shard **al azar**
- El orden no importa: no hace falta shardear por key
- Para borrar hay que saber el shard: el mensaje lo devuelve, **opaco**
- Explicación posible del `ReceiptHandle`
- Una lectura inmediata acierta el shard **1 de cada 3** veces: consistencia eventual

Opción 3: sharding con replicación

- El shard que recibe el mensaje lo guarda y lo copia a **dos shards al azar**
- Cada copia guarda también la **lista de hosts**: `MSG, [H1, H2, H3]`
- Sin líder, sin términos, sin votaciones, sin quórum
- Escala agregando shards; la replicación se configura
- Precio: un mismo mensaje puede leerse desde **cada réplica**

Opción 3.1: solo responde el primary

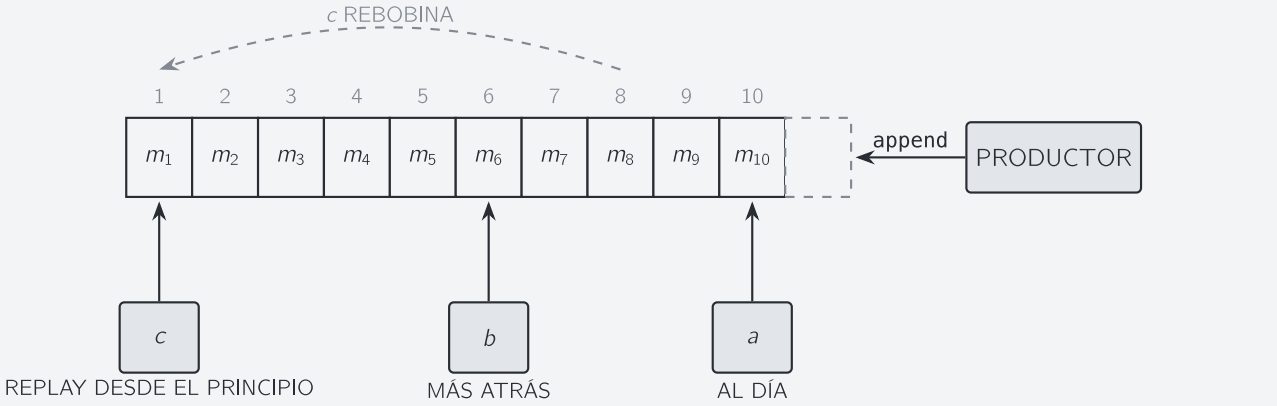
- El nodo que recibe la escritura es el **primary**; elige un **secondary** y un **tertiary**
- Solo el primary devuelve el mensaje en un `receive`
- Detección de fallas por **gossip** (o un servicio como ZooKeeper)
- Si el primary cae, el secondary pasa a responder
- El visibility timeout es **local**: tras una falla, el mensaje se **duplica**

EVENT STREAMING Y LOGS PARTICIONADOS

Log persistente

- En los dos modelos anteriores los mensajes son **efímeros**: son sistemas de transporte
- En un **log**, los mensajes quedan guardados y **ordenados**
- Hay un **período de retención**: una semana por omisión, configurable a meses o sin límite
- Caso de uso: streams de eventos, como el **engagement** de una red social

Replay y fan-out trivial



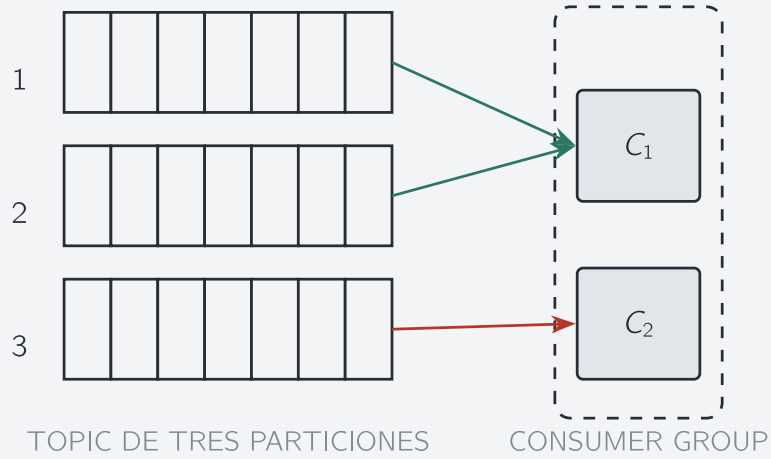
- Cada consumidor lee desde su propia posición: el **offset**
- Puede empezar desde cualquier lado, **rebobinar** y hacer **replay**
- El fan-out es trivial: leer no destruye el mensaje

Particionado por key

- `send(topic, key, payload)`: el topic tiene **N particiones**
- La partición sale de `hash(key) % N`
- Garantía: **FIFO por partición**; no hay orden global del topic
- La key representa el negocio: con el user ID, cada usuario queda ordenado

Consumer groups

- Cada partición la lee **un solo** consumidor del grupo
- Un consumidor puede leer **varias** particiones
- Conviene tener **más particiones que consumidores**
- Un **consumer group** es una aplicación: sus procesos se reparten el topic



Coordinador

- Asigna particiones a los consumidores y las **rebalancea** cuando el grupo cambia
- Los consumidores le mandan **checkpoints** de su offset
- Primera versión: **ZooKeeper** más lógica en el cliente
  - Nodos efímeros para tomar una partición; watches para enterarse de cambios
- Hoy el coordinador está en Kafka, y ZooKeeper se reemplazó por **KRaft**

Tolerancia a fallas, para leer: Jun Rao, «Intra-cluster Replication in Apache Kafka», LinkedIn Engineering, 2013

Síntesis

- **Producer-consumer**: un consumidor por mensaje; balanceo de carga
- **Publisher-subscriber**: el mensaje se multiplica; fan-out
- **Event streaming**: un log persistente, particionado, que cada uno lee a su ritmo
- Los brokers son **sistemas de storage** especializados en soltar mensajes