Паттерн Transactional Outbox

В микросервисах часто нужно сделать два действия одновременно: сохранить состояние (например, «Заказ создан») в базу данных и уведомить другие сервисы через брокер сообщений (Kafka/RabbitMQ). Если делать это наивно, возникают две фатальные ситуации:

1. **Сначала база, потом брокер:** База сохранила заказ, но брокер упал или сеть моргнула. Результат: Заказ есть, но склад и доставка о нем не знают. Система несогласована.
2. **Сначала брокер, потом база:** Сообщение ушло, но транзакция базы откатилась (например, из-за нарушения constraint'а). Результат: Склад собирает несуществующий заказ.

Решение: Transactional Outbox Паттерн предлагает использовать надежность локальных **ACID-транзакций** базы данных вместо сложных и медленных распределенных транзакций (2PC).

Механика работы Transactional Outbox

Суть паттерна проста: мы превращаем отправку сообщения в операцию записи в базу данных. Вместо того чтобы координировать две ненадежные системы (БД и Брокер), мы используем надежность локальных **ACID-транзакций**.

1. Атомарная запись Когда вы создаете заказ, вы делаете `INSERT` сразу в две таблицы в рамках одной транзакции:

1. Таблица `Orders` (Бизнес-данные).
2. Таблица `Outbox` (Намерение отправить сообщение).

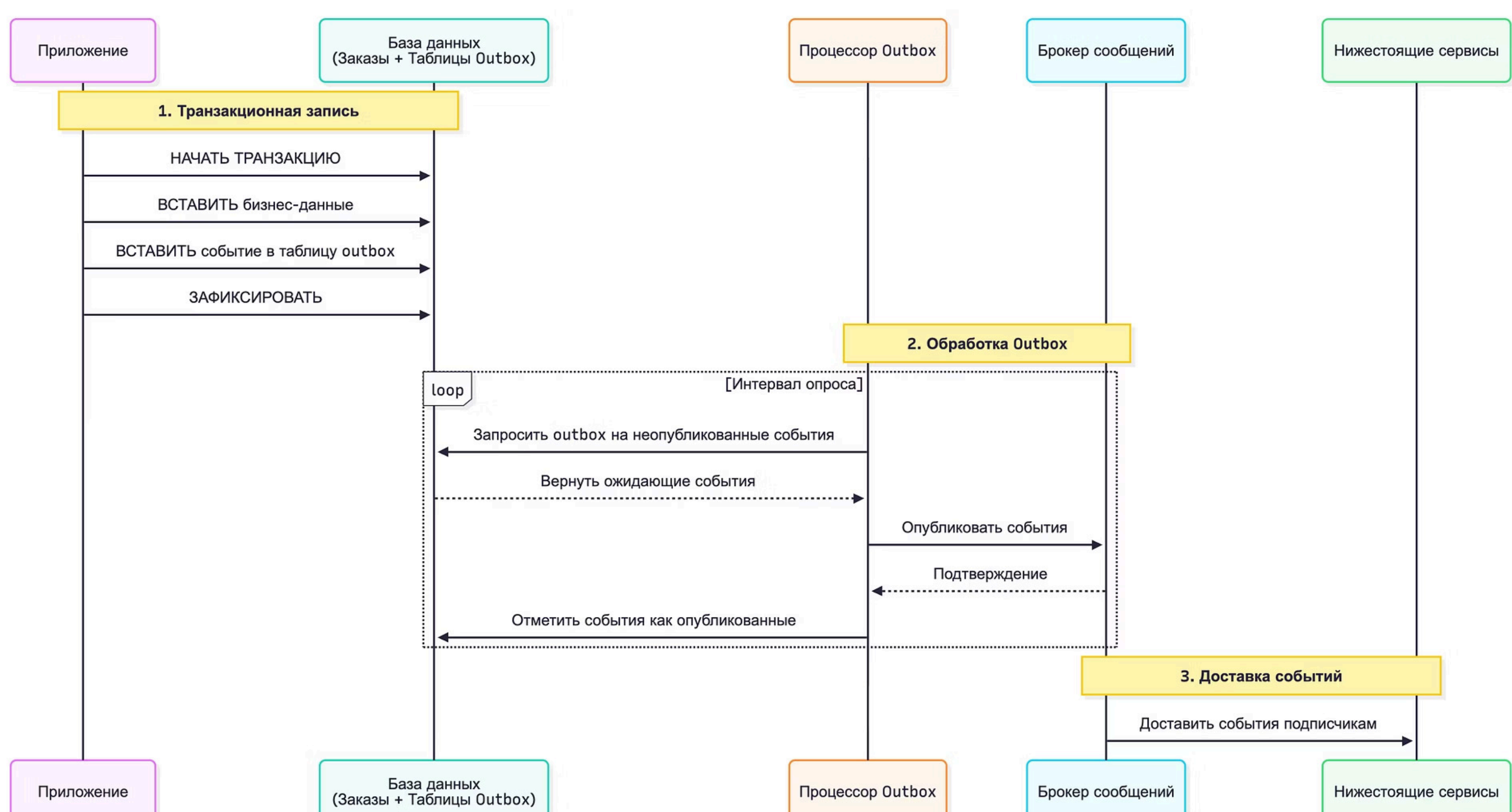
Если транзакция проходит (Commit), гарантированно сохраняются обе записи. Если происходит сбой (Rollback) — не сохраняется ни одна. Состояние «Заказ есть, а событие потерялось» становится невозможным.

2. Структура таблицы Outbox Обычно эта схема выглядит так:

- `id`: Уникальный идентификатор сообщения (UUID).
- `aggregate_id`: ID сущности (например, ID заказа), чтобы соблюдать порядок событий при чтении (sharding key).
- `payload`: JSON с данными события.
- `topic`: Название топика в Kafka/RabbitMQ.
- `created_at`: Время создания.
- `processed_at`: Время отправки (изначально `NULL`).

3. Доставка сообщений (The Relay) Записать данные — это полдела. Теперь их нужно извлечь из таблицы и отправить в брокер. Есть два основных подхода:

- **Polling Publisher (Опрос):** Классический и надежный метод. Фоновый процесс (воркер) периодически (например, раз в секунду) делает SQL-запрос: `SELECT * FROM outbox WHERE processed_at IS NULL ORDER BY created_at LIMIT 50`. Он отправляет пачку событий в брокер и, после получения подтверждения (ACK), обновляет поле `processed_at`.
  - *Плюсы:* Работает с любой базой, легко отлаживать, не требует сложной инфраструктуры.
  - *Минусы:* Небольшая задержка (Polling Delay), лишняя нагрузка на базу данных от постоянных SELECT'ов.
- **Log Tailing (CDC - Change Data Capture):** Продвинутый подход (например, с использованием **Debezium**). Специальный коннектор читает журнал транзакций базы данных (Write-Ahead Log в Postgres или Binlog в MySQL) и стримит изменения таблицы `outbox` в брокер напрямую.
  - *Плюсы:* Минимальная задержка (Real-time), не нагружает базу SELECT-запросами.
  - *Минусы:* Высокая сложность эксплуатации. Привязка к внутренней реализации конкретной СУБД (Vendor Lock-in).



Альтернатива: Транзакции в Apache Kafka

Kafka предлагает свое решение проблемы двойной записи, но с важной оговоркой: оно работает только внутри экосистемы Kafka. Это реализуется через **Transactions API** и обеспечивает семантику **Exactly-Once** (Ровно один раз).

Паттерн Consume-Process-Produce Этот подход идеален для стриминговой обработки, когда ваше приложение читает данные из одного топика (Source), обрабатывает их и пишет результат в другой топик (Sink).

Как это работает: Kafka позволяет объединить два действия в одну атомарную транзакцию:

1. Отправку новых сообщений в выходной топик.
2. Фиксацию смещения (Commit Offset) в входном топике (пометка сообщения как «обработанного»).

Если транзакция прерывается, оффсет не коммитится, и сообщение не появляется в выходном топике (для consumers с уровнем изоляции `read_committed`). Это гарантирует, что сообщение не будет потеряно и не будет обработано дважды.

Ограничения Это **не заменяет** классический Outbox паттерн для большинства микросервисов.

- *Где работает:* Если ваш State Store находится внутри Kafka (например, Kafka Streams или KSQL).
- *Где НЕ работает:* Если вам нужно обновить внешнюю базу данных (Postgres, MySQL) и отправить сообщение. Транзакции Kafka не могут управлять коммитом в PostgreSQL. Это разные миры, которые не могут быть объединены в одну распределенную транзакцию без тяжелых протоколов вроде 2PC (XA), от которых современная индустрия старается уходить.

По сути, это специализированная форма Outbox, где роль базы данных и роль брокера выполняет одна система — Kafka.

Суровая реальность эксплуатации (Operational Reality)

Паттерн Outbox выглядит элегантно на бумаге, но в продакшене вы столкнетесь с инженерными вызовами, которые редко упоминаются в учебниках.

1. Проблема порядка (Event Ordering) Порядок вставки (`INSERT`) не гарантирует порядок доставки.

- **Гонка процессов:** Если у вас несколько инстансов Relay-процесса (воркеров), они могут вычитывать события параллельно. Сетевые задержки могут привести к тому, что событие «Обновление заказа» придет потребителю раньше, чем событие «Создание заказа».
- **Решение:**
  - Использовать **Sequence ID** (глобальный счетчик) или монотонно возрастающие Timestamp.
  - Использовать **Partition Keys** (ключи партицирования) в Kafka. Все события одного `OrderID` должны попадать в одну партицию, чтобы гарантировать последовательную обработку.

2. Раздувание таблицы (Table Bloat & Cleanup) Таблица `Outbox` — это таблица с высокой интенсивностью записи и удаления (High Churn).

- **Проблема:** Если вы просто удаляете обработанные строки (`DELETE`), в базах вроде PostgreSQL это приводит к фрагментации и раздуванию таблицы (Table Bloat), что убивает производительность.
- **Стратегия:**
  - **Soft Delete:** Помечать как `processed`, но не удалять сразу. Удалять пачками ночью.
  - **Table Partitioning:** Лучшее решение. Создавать новую партицию таблицы каждый день/час. Старые партиции с обработанными событиями можно удалять мгновенно (`DROP PARTITION`), что гораздо дешевле, чем `DELETE` миллионов строк.

3. Дубликаты и Идемпотентность Outbox гарантирует доставку **At-Least-Once** (минимум один раз), но не **Exactly-Once**.

- **Сценарий:** Relay отправил сообщение в брокер, но упал до того, как успел обновить статус в БД (`UPDATE outbox SET processed = true`). После перезапуска он отправит то же сообщение снова.
- **Императив:** Ваши потребители (Consumers) **обязаны быть идемпотентными.** Они должны уметь распознавать и игнорировать дубликаты (например, проверяя `event_id` в Redis или дедуплицируя записи на уровне БД).

4. Обработка сбоев (Error Handling) Что если брокер лежит?

- **Backoff Strategy:** Не долбите брокер в бесконечном цикле. Используйте экспоненциальную задержку (Exponential Backoff).
- **Poison Messages:** Если сообщение не может быть сериализовано (битые данные), оно заблокирует всю очередь. Такие сообщения нужно отбрасывать в **DLQ (Dead Letter Queue)** после N неудачных попыток, чтобы не останавливать конвейер.
- **Мониторинг:** Вам нужны алерты не только на падение Relay, но и на **Lag** (отставание). Если в таблице Outbox скапливаются события быстрее, чем вы их отправляете, система деградирует.

Почему это работает: Философия паттерна

Успех Transactional Outbox строится на том, что он решает сложную проблему распределенных систем локальными средствами.

1. Опора на ACID, а не на сеть Вместо того чтобы строить хрупкие распределенные транзакции (Distributed Transactions) между сервисами, мы используем то, что базы данных умеют делать лучше всего — атомарно сохранять данные. Мы превращаем сетевую проблему в проблему локального хранения.

2. Временная развязка (Temporal Decoupling) Это самое важное архитектурное преимущество.

- **Без Outbox:** Если Kafka упала, вы не можете создать заказ. Ваша доступность жестко связана с доступностью брокера.
- **С Outbox:** Вы продолжаете принимать заказы, даже если Kafka лежит уже 2 часа. События просто копятся в таблице. Как только связь восстановится, Relay доставит их. Ваша бизнес-логика изолирована от проблем инфраструктуры.

3. Бесплатный Аудит (Audit Trail) Таблица `Outbox` — это хронологическая лента всех изменений в системе. До тех пор, пока вы не очистите старые записи, у вас есть идеальный лог для отладки: вы точно знаете, *что* произошло, *когда* и *было ли* это отправлено.

4. Прагматизм важнее элегантности Да, опрос таблицы (Polling) или настройка CDC кажется «костылем» по сравнению с красивыми диаграммами прямой отправки сообщений. Но в продакшене **надежность бьет элегантность**. Outbox — это скучное, проверенное решение, которое гарантирует целостность данных там, где другие подходы теряют транзакции.

Мы полностью закрыли тему **Transactional Outbox**. Это был глубокий и технически насыщенный модуль.

Что дальше? У нас есть несколько путей для продолжения курса System Design:

1. **Idempotency (Идемпотентность):** Логичное продолжение. Мы сказали, что Consumer должен быть идемпотентным — пора объяснить, *как* это сделать.
2. **Distributed Sagas (Саги):** Как управлять транзакцией, которая охватывает 5 микросервисов (если Outbox решает проблему только одного шага).
3. **CQRS:** Разделение моделей чтения и записи (часто идет в паре с Event Sourcing и Outbox).