Семантика «точно один раз»

Обработка сообщений «точно один раз» решает фундаментальную проблему распределенных систем: обеспечение обработки сообщений ровно один раз, избегая как потери сообщений (не более одного раза), так и дублирования обработки (не менее одного раза). Современные системы обмена сообщениями, такие как Kafka, перенесли эту сложность на уровень инфраструктуры.

В данном анализе рассматривается реализация семантики «точно один раз» в компонентах системы, что обеспечивает техническую основу для реализации аналогичных гарантий в распределенных архитектурах.

Основная проблема: почему сложно реализовать «точно один раз»

Фундаментальная проблема связана с распределенным характером систем обмена сообщениями. Рассмотрим следующую последовательность:

01

Производитель отправляет сообщение брокеру

02

Брокер успешно сохраняет сообщение

03

Сеть выходит из строя до того, как брокер может подтвердить получение

04

Производитель предполагает сбой и повторяет попытку

05

Результат: дублирование сообщения

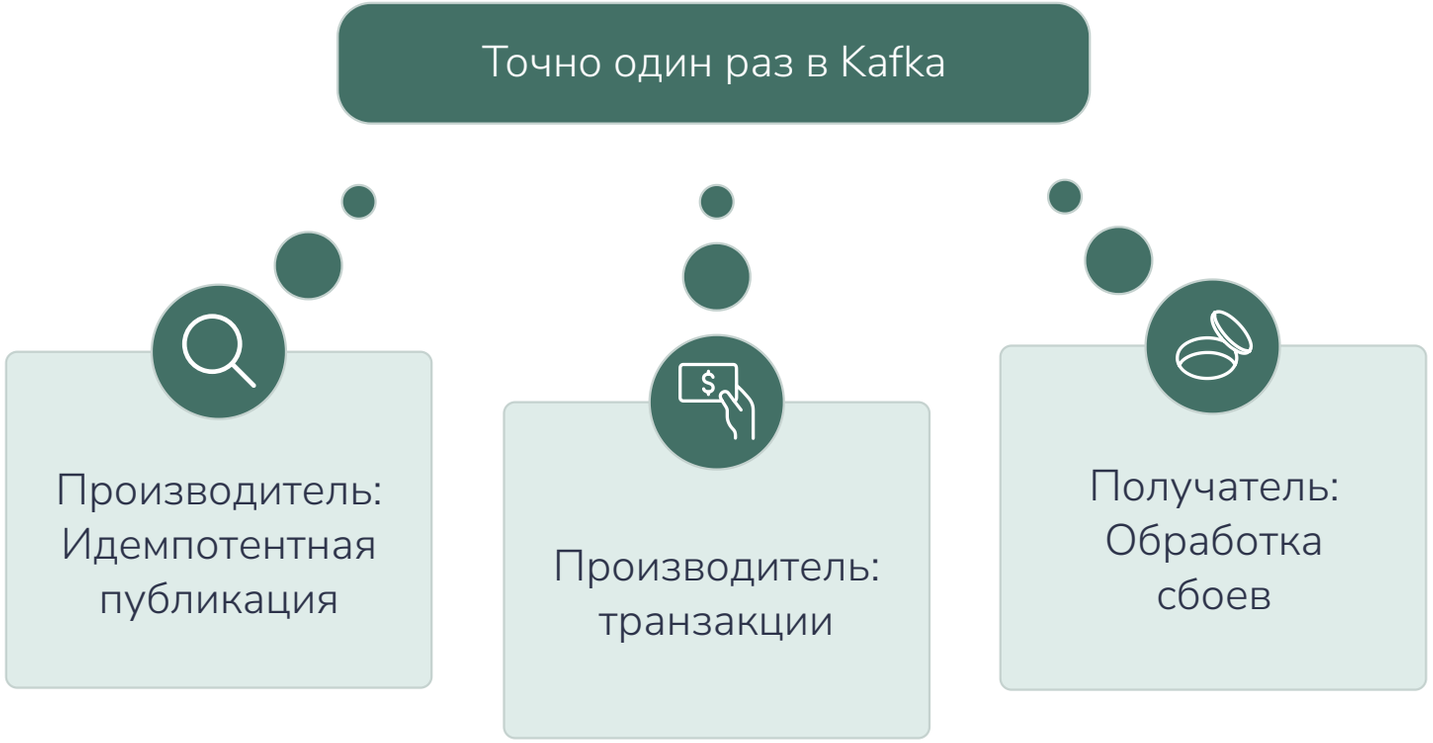
Традиционные подходы требовали сложности на уровне приложения:

- **Ручная дедупликация**: отслеживание обработанных идентификаторов сообщений в базе данных
- **Идемпотентные операции**: бизнес-логика, предназначенная для обработки дубликатов
- **Двухфазная фиксация**: межсистемная координация транзакций с затратами на производительность

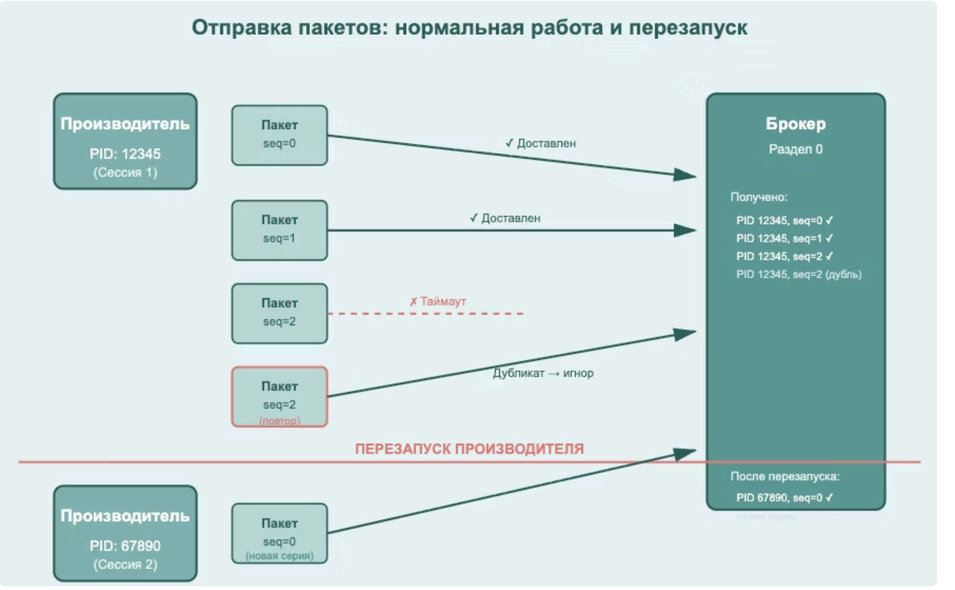
Современные системы обмена сообщениями решают эту проблему на уровне инфраструктуры с помощью предсказуемых, повторно используемых механизмов. Ниже мы опишем алгоритм того, как это реализовано в **Kafka.**

Точно один раз в Kafka

Семантика «точно один раз» достигается с помощью двух взаимодополняющих механизмов на стороне производителя: **идемпотентной публикации** для операций с одним разделом и **транзакционной публикации** для операций с несколькими разделами. Но даже этого не всегда достаточно: если приложение упадет до того как подтвердит получение, возникнет дупликация. Подробно рассмотрим все три темы ниже.



Сторона производителя: PID и порядковые номера



Посмотрите на диаграмму выше. Слева вы видите производителя с надписью **PID: 12345**. Это **идентификатор производителя** (Producer ID) — уникальный номер, который брокер Kafka выдает при запуске. Пока производитель работает, этот номер остается неизменным и позволяет брокеру отслеживать, от кого пришли сообщения.

Справа от производителя вы видите пакеты с надписями **seq=0**, **seq=1**, **seq=2**. Это **порядковые номера** (sequence numbers) — счетчик отправок, который увеличивается с каждым новым пакетом сообщений.

Как работает нумерация

В Kafka сообщения отправляются не по одному, а **пакетами (batch)** для эффективности. Каждый пакет получает свой порядковый номер:

- **seq=0** — первый пакет
- **seq=1** — второй пакет
- **seq=2** — третий пакет
- и так далее...

Защита от дубликатов

Теперь посмотрите на третий пакет на диаграмме (**seq=2**). Обратите внимание: он отправляется **дважды**, т.к. подтверждение о получении сообщение не было доставлено производителю, и он решает сделать повторную попытку.

Что происходит:

1. **Первая попытка (пунктирная красная линия)** — производитель отправляет пакет seq=2, брокер успешно его получает и сохраняет. Но ответ брокера теряется в сети из-за таймаута. Производитель не получил подтверждение и думает: "Отправка провалилась!"
2. **Повторная попытка (сплошная зеленая линия)** — производитель автоматически отправляет пакет еще раз. **Критический момент:** это тот же самый пакет с тем же номером seq=2, потому что номер был присвоен при создании пакета, а не при отправке.
3. **Брокер распознает дубликат** — когда пакет seq=2 приходит второй раз, брокер проверяет: "Стоп, я уже обработал seq=2 от этого производителя (PID=12345)!" и молча игнорирует его. На диаграмме это показано надписью **"Дубликат → игнор"**.

Справа в секции брокера видно, как он отслеживает получение: seq=0 ✓, seq=1 ✓, seq=2 ✓, а затем seq=2 (дубль) — отброшен.

Перезапуск меняет все

Теперь посмотрите на **красную сплошную линию** посередине диаграммы с надписью "ПЕРЕЗАПУСК ПРОИЗВОДИТЕЛЯ".

После перезапуска производитель получает **новый PID: 67890** (вместо старого 12345). Для брокера это теперь совершенно другой производитель. Поэтому:

- Нумерация начинается заново с **seq=0**
- Это НЕ дубликат первого пакета из верхней части (хотя номер тот же)
- Брокер воспринимает это как **новую независимую серию** сообщений

На диаграмме это показано надписью "(новая серия)" под последним пакетом.

Как брокер проверяет дубликаты

Для каждой комбинации **(PID + раздел)** брокер помнит: "Какой seq я обработал последним?" При получении нового пакета он сравнивает:

- **seq меньше или равен последнему** → дубликат, отбрасываем (как seq=2 во второй раз на диаграмме)
- **seq на 1 больше последнего** → новый пакет, обрабатываем (как seq=0, seq=1, первый seq=2)
- **seq намного больше** → пропущены сообщения, ошибка

Именно поэтому на диаграмме все пакеты с галочками ✓ успешно обработаны, а повторный seq=2 отброшен — брокер видит, что уже получал этот номер от этого производителя.

Зомби

Помните на диаграмме, что после перезапуска производитель получил **новый PID: 67890**? Это создает серьезную проблему: брокер больше не может связать новые сообщения со старыми. Нумерация начинается с нуля, и все гарантии идемпотентности между сеансами работы теряются.

Но есть еще хуже сценарий — **проблема зомби-производителя**. Представьте ситуацию:

1. ### Стартует производитель

   Производитель работает и отправляет сообщения

2. ### Сбой сети

   Происходит **сбой сети** — производитель теряет связь с брокером

3. ### Стартует новый производитель

   Система мониторинга думает, что производитель умер, и **запускает новый экземпляр**

   Но старый производитель на самом деле жив! Сеть восстановилась, и он продолжает отправлять сообщения

4. ### Зомби

   Теперь **два производителя** отправляют сообщения одновременно, один из них - "зомби"

Зомби: решение

Kafka решает обе проблемы (потеря связи при перезапуске + зомби) с помощью двух механизмов:

1. Транзакционный ID (transactional.id) Это настраиваемый пользователем строковый идентификатор, который представляет логическую идентичность службы. Например:

"payment-service" — для всех экземпляров сервиса платежей "order-processor-region-eu" — для процессора заказов в Европе "analytics-pipeline-stage-1" — для первой стадии аналитического конвейера

1. Номер эпохи (epoch) Это отдельный 16-битный счетчик (не путать с PID!), который увеличивается при каждом перезапуске производителя с тем же transactional.id.

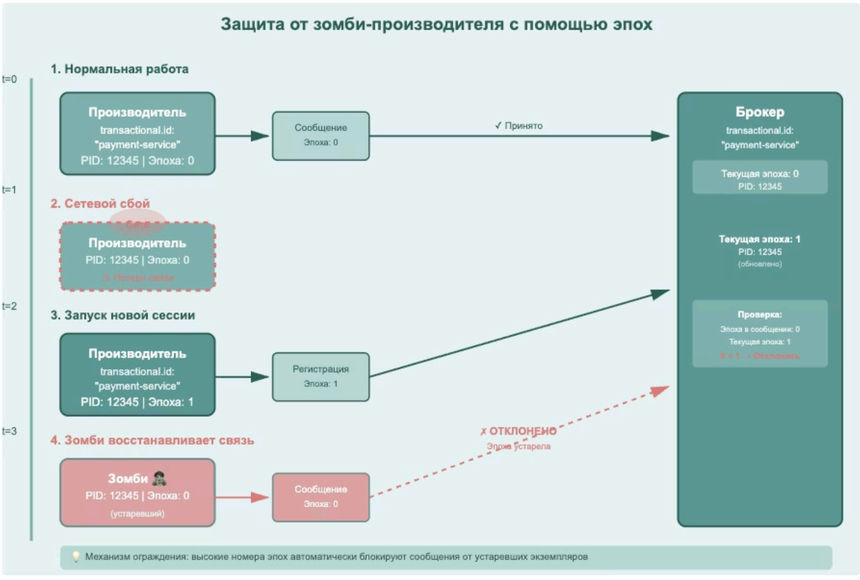
Когда производитель настроен с transactional.id, происходит следующее:

```
Сессия 1 (первый запуск)
 ├── transactional.id: "payment-service"
 ├── PID: 12345  ← остается неизменным!
 ├── Эпоха: 0
 └── Отправляет сообщения seq=0, 1, 2...

Сбой или перезапуск
 └── Производитель перезапускается

Сессия 2 (после перезапуска)
 ├── transactional.id: "payment-service"  ← тот же
 ├── PID: 12345  ← тот же самый!
 ├── Эпоха: 1  ← увеличилась!
 └── Продолжает seq=3, 4, 5...
```

Ключевое отличие от нашей диаграммы выше: С transactional.id PID **НЕ меняется** при перезапуске! Это позволяет сохранить непрерывность отслеживания последовательности.

Как это решает проблему зомби

Высокие номера эпох автоматически **"ограждают"** (fence out) предыдущие экземпляры. Зомби не может навредить, потому что его сообщения игнорируются.



Вспомните первую диаграмму: после перезапуска производитель получил совершенно новый PID (67890 вместо 12345). Брокер воспринял это как нового производителя, и вся история отправок была потеряна.

Что изменилось с transactional.id:

Посмотрите на **этап 3** новой диаграммы. После перезапуска:

- `PID` остается **тем же самым: 12345** (не меняется!)
- `transactional.id` тот же: `"payment-service"`
- Только **эпоха увеличилась**: 0 → 1

Брокер теперь **знает**, что это тот же логический производитель, просто новая сессия. Непрерывность сохраняется.

Защита от зомби (этап 4):

Старый производитель (зомби, эпоха 0) пытается отправить сообщение. Брокер проверяет в правой панели:

```
Эпоха в сообщении: 0
Текущая эпоха: 1
0 < 1 →    ОТКЛОНЕНО
```

Два ключевых улучшения:

1. **Сохранение PID** решает проблему потери истории — брокер продолжает отслеживать последовательность seq между перезапусками
2. **Механизм эпох** решает проблему зомби — старые экземпляры автоматически блокируются, даже если они имеют правильный PID

Теперь у нас есть и непрерывность (один PID), и безопасность (эпохи ограждают зомби).

Транзакционная публикация: атомарность нескольких разделов

Почему одной идемпотентности недостаточно

Мы только что разобрали, как PID, порядковые номера и эпохи обеспечивают **идемпотентность** — гарантию, что одно сообщение не обработается дважды. Но это работает только **внутри одного раздела**.

Идемпотентность (PID + seq + epoch) защищает от дубликатов **в одном разделе**, но не гарантирует **атомарность между разделами**. Для этого нужны транзакции.

Что дают транзакции

Транзакции Kafka обеспечивают два критических свойства:

1. **Атомарность между разделами** — либо все сообщения доставлены во все разделы, либо ни одно
2. **Идемпотентность на уровне всей операции** — повторная попытка транзакции не приведет к дублированию

Именно поэтому `transactional.id` это второе обязательный механизм, который гарантирует доставку точно один раз.

Координация транзакций

Для операций, охватывающих несколько разделов или тем, производители используют **транзакционные API**. Производитель координирует свои действия с **координатором транзакций**, чтобы обеспечить атомарность фиксации во всех задействованных разделах.

Транзакции используют гарантии «точно один раз» в отдельных разделах, чтобы обеспечить атомарность в нескольких разделах и гарантии «точно один раз». Это достигается с помощью распределенных транзакций, но ограничено брокерами Kafka в пределах одного кластера — не распространяется на внешние системы, такие как базы данных.

Поток транзакций с точки зрения производителя:

Начало транзакции

получение идентификатора транзакции от координатора

Отправка сообщений

в несколько разделов/тем (все помеченные как часть этой транзакции) — гарантируется, что они будут отправлены ровно один раз

Фиксация транзакции

сообщение координатору о необходимости атомарной фиксации во всех разделах

Обработка ответа координатора

успех означает, что все разделы зафиксированы, сбой означает, что все разделы отменены

Транзакции могут охватывать совершенно разные темы, а не только разделы в пределах одной темы. Это позволяет осуществлять атомарные сложные многосервисные операции.

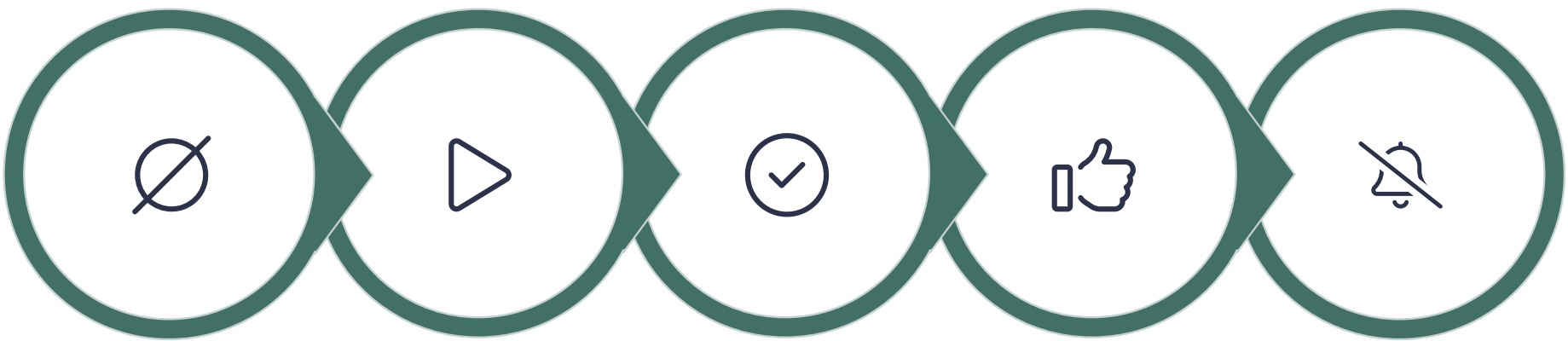
Сторона брокера: проверка, координация и хранение

С точки зрения брокера, семантика «точно один раз» требует поддержания дополнительного состояния и реализации сложной логики проверки.

Координация транзакций

Координатор транзакций — это специальный компонент брокера, который управляет жизненным циклом многораздельных транзакций. Он поддерживает состояние транзакций в реплицированной внутренней теме под названием __transaction_state.

Прогресс состояния транзакции:



Реализация двухфазного коммита

Координатор транзакций использует классический протокол двухфазной фиксации (2PC), адаптированный для распределенной природы Kafka. Рассмотрим, как это работает на практике.

Фаза 1 (подготовка)

координатор записывает «PrepareCommit» в журнал транзакций

Фаза 2 (фиксация)

координатор записывает маркеры фиксации во все участвующие разделы

Завершение

координатор записывает «CompleteCommit» в журнал транзакций

Координатор должен обрабатывать случай, когда координатор транзакций сам выходит из строя в середине транзакции. Реплицированный журнал транзакций позволяет координаторам восстановления завершить или прервать незавершенные транзакции.

⚠ Двухфазный коммит гарантирует, что либо все сообщения транзакции станут видимыми во всех разделах одновременно (атомарная фиксация), либо ни одно из них не станет видимым (атомарная отмена), даже при сбоях координатора в процессе выполнения.

Дедупликация транзакций

Здесь брокер реализует важную идею, что **атомарность сама по себе не обеспечивает точное выполнение**. Даже если транзакция фиксируется атомарным образом во всех разделах, производитель может не получить подтверждение успеха и повторить всю транзакцию.

Проверка дедупликации: когда производитель пытается зафиксировать транзакцию, координатор проверяет:

Проверка дублирования

- Уже ли эта сессия производителя (PID + Epoch) зафиксировала транзакцию с этой последовательностью?
- Если да: вернуть успех без повторного выполнения (идемпотентность)
- Если нет: продолжить обычную двухфазную фиксацию (атомарность)

> Именно эта комбинация идемпотентности на уровне транзакций и атомарных фиксаций обеспечивает истинную семантику «точно один раз».

Контрольные записи и изоляция потребителей

Для потребителей, настроенных с помощью isolation.level=read_committed, брокеры реализуют дополнительную логику:

Последний стабильный смещение (LSO)

потребители могут читать только до смещения первого сообщения, принадлежащего открытой (незафиксированной) транзакции. Это предотвращает чтение сообщений, которые могут быть впоследствии прерваны.

Фильтрация контрольных записей

брокеры отфильтровывают прерванные транзакционные сообщения перед отправкой их потребителям. Потребители никогда не видят сообщения из прерванных транзакций.

Маркеры транзакций

специальные записи управления указывают границы транзакций (фиксация/прерывание), но не отображаются в приложениях в виде обычных сообщений.

Сторона потребителя: обработка сообщений ровно один раз

С точки зрения потребителя, семантика «ровно один раз» касается не только доставки сообщений, но и обеспечения атомарности обработки сообщений и обновления состояния.

Задача потребителя

Даже при точном однократном доставке от брокера потребители сталкиваются со своей собственной задачей:

1. Прием и обработка сообщений
2. Обновление состояния приложения (база данных, кэш и т. д.)
3. Фиксация смещений потребителя

Сценарий сбоя: если потребитель выходит из строя после шага 2, но до шага 3, он повторно обработает те же сообщения, не зная, что они уже были обработаны.

Обработка потребителем

Kafka обеспечивает семантику «точно один раз» от начала до конца (производитель → брокер → потребитель) при правильной настройке. Однако потребители все равно могут видеть дубликаты сообщений, если они не управляют смещениями правильно — это проблема настройки и использования, а не ограничение гарантий «точно один раз» Kafka. Мы подробнее остановимся на этом ниже.

Как работает доставка «точно один раз» со стороны потребителя:

- Потребители с isolation.level=read_committed видят только сообщения из зафиксированных транзакций
- Брокер отслеживает смещения потребителей и доставляет только те сообщения, которые еще не были подтверждены
- Когда смещения зафиксированы (вручную или автоматически), эти сообщения не доставляются повторно
- Если потребитель выходит из строя до фиксации смещений, он получит те же сообщения снова

Возможно, вы уже задаетесь вопросом: что произойдет, если приложение завершит работу после фиксации изменений в бизнесе (например, снятие средств со счета), но до подтверждения сообщения Kafka? Мы можем получить худший результат: изменение в бизнесе произошло, но Kafka повторно доставит сообщение (при перезапуске приложения), что может привести к дублированию обработки (двойному снятию средств).

Даже при такой доставке «точно один раз» между обработкой сообщений и фиксацией смещений остается зазор, который может привести к дублированию обработки. Существует несколько решений этой проблемы:

Транзакционный подход Kafka

вместо того, чтобы выполнять изменения в коде напрямую, мы можем делать это асинхронно: отправлять сообщение другому компоненту, который выполнит это неидентичное изменение. Мы можем использовать транзакции Kafka для координации фиксации смещений (подтверждения, что сообщение прочитано) с созданием этого бизнес-сообщения. Это сделает все операции в Kafka атомарными.

Ручное управление смещениями

храните смещения потребителей в той же системе, что и состояние вашего бизнеса (например, в базе данных), и фиксируйте их вместе с помощью внешних транзакций.

Подход с использованием идемпотентности

разработайте логику обработки так, чтобы безопасно обрабатывать дубликаты сообщений, сделав систему устойчивой к повторной обработке независимо от времени смещения.

Вопросы производительности и эксплуатации

Семантика «точно один раз» не бесплатна. Вот что вы на самом деле платите за эти гарантии:

2-5%	10-20%
Снижение пропускной способности	Замедление транзакций
идемпотентные производители добавляют небольшую накладную накладную (в основном за счет учета порядковых номеров)	полные транзакции могут замедлить работу. Эта координация и двухфазный процесс фиксации занимают время

Затраты на память: брокеры должны отслеживать порядковые номера для каждой активной комбинации производителя и раздела, хотя обычно это не занимает много места. Более серьезной проблемой является необходимость буферизации потребителями сообщений из текущих транзакций — при длительных транзакциях это может быстро занять много памяти.

Тонкости настройки: нельзя просто переключить переключатель и получить точно однократное выполнение. Идемпотентность производителя требует, чтобы max.in.flight.requests был равен 5 или меньше для поддержания порядка сообщений. Таймауты транзакций сложно настроить — слишком короткие приводят к прерыванию легитимных операций, слишком длинные — к блокировке потребителей неработающими производителями. Кроме того, необходимо использовать acks=all, иначе гарантии долговечности теряются.

Вывод: семантика «точно один раз» добавляет сложность и накладные расходы, но для систем, где дубликаты или потери создают реальные проблемы для бизнеса, это обычно оправдано.

Как другие системы обмена сообщениями решают эту проблему

Kafka — не единственная система на рынке, но она определенно лидирует, когда речь идет о семантике «точно один раз». Вот как выглядят конкуренты:

Многие современные системы также поддерживают точно однократное выполнение:

Apache Pulsar

использует более простой подход — автоматическую дедупликацию на уровне брокера с использованием идентификаторов сообщений. Это менее сложно, чем система Kafka, но гораздо проще в использовании. Вы получаете базовую точность однократного выполнения без сложной настройки.

NATS JetStream

предлагает гибкие окна дедупликации, которые можно настраивать, включая «бесконечную дедупликацию», если вы не хотите видеть одно и то же сообщение дважды. Это мощный инструмент, но требует более тщательной настройки.

Amazon Kinesis

вероятно, ближайший конкурент Kafka, но он не обеспечивает полную точность доставки. Он имеет порядковые номера для каждого фрагмента (аналогично подходу Kafka для каждой партиции) и обеспечивает доставку не реже одного раза, но вам все равно нужно реализовать собственную логику дедупликации в приложениях.

Amazon SQS FIFO

обеспечивает дедупликацию с помощью идентификаторов дедупликации сообщений — вы либо предоставляете свой собственный идентификатор, либо AWS генерирует его на основе содержимого сообщения. Но есть один нюанс: это работает только в течение 5-минутного окна и ограничено одной очередью. Подходит для простых случаев использования, но не для сложных распределенных операций.

Amazon SNS FIFO

работает аналогично SQS FIFO — использует идентификаторы дедупликации для предотвращения дублирования публикаций в течение 5 минут. Однако это только предотвращает публикацию дубликатов сообщений в теме; не обеспечивает атомарность между темами, как транзакции Kafka.

Системы, которые обходят эту проблему:

RabbitMQ и **ActiveMQ** в основном сдаются и говорят: «Разберись сам». Они обеспечивают доставку «хотя бы один раз» и ожидают, что вы будете обрабатывать дедупликацию в коде приложения. Это старый подход, который перекладывает всю сложность на вас.

> **Вывод**: большинство систем предоставляют либо базовую дедупликацию, либо атомарные операции, но не и то, и другое. Инновация Kafka заключалась в том, что для реального решения проблемы «точно один раз» необходимо сочетание идемпотентности и атомарности.

Когда применять семантику «точно один раз»

Семантика «точно один раз» обеспечивает высокую согласованность, но влечет за собой сложность и снижение производительности. Сценарии применения требуют тщательной оценки.

Критические случаи использования, требующие гарантий «точно один раз»:

- **Финансовые транзакции**: обработка платежей, торговые системы, бухгалтерский учет, где дублирование обработки приводит к денежным потерям
- **Управление запасами**: уровни запасов в электронной коммерции, системы цепочки поставок, где двойная обработка приводит к перепродаже
- **Системы обеспечения соответствия**: медицинские записи, финансовая отчетность, системы аудита, требующие нормативной точности

Случаи использования, в которых точность выполнения может быть ненужной:

- **Аналитика и метрики**: крупномасштабная обработка данных, где незначительные отклонения в точности не влияют на бизнес-решения
- **Операционный мониторинг**: системные журналы, метрики производительности, где низкая задержка имеет приоритет над идеальной точностью
- **Некритические уведомления**: уведомления по электронной почте, обмен сообщениями, где дублирование доставки представляет собой приемлемое неудобство

Ключевые выводы

1. Exactly-once требует как идемпотентности, так и атомарности

   ни один из этих механизмов в отдельности не является достаточным

2. Номера последовательности производителя

   обеспечивают основу для устранения дубликатов в разделах

3. Координация транзакций

   распространяет гарантии exactly-once на несколько разделов и тем

4. Важность дизайна потребителя

   даже доставка exactly-once требует атомарной обработки и управления смещением

5. Компромиссы в производительности реальны

   точно однократная доставка сопровождается измеримыми накладными расходами в пропускной способности и задержках

6. Решения на уровне инфраструктуры

   масштабируются лучше, чем перенос сложности на каждое приложение

Понимание этих механизмов дает вам инструменты для оценки систем обмена сообщениями, проектирования отказоустойчивых приложений и реализации аналогичных гарантий в вашей собственной архитектуре распределенных систем.