



Онлайн-образование

Меня хорошо видно && слышно?

Ставьте + , если все хорошо
Напишите в чат, если есть проблемы

Распределённые очереди сообщений Kafka и RabbitMQ



Зайгин Вадим

Senior Data Engineer

VMware

vzaigrin@yandex.ru

Вадим Заигрин



vzaigrin@yandex.ru

Senior Data Engineer

vmware®

Более 30 лет в ИТ:

- Big Data
 - Data Engineer
 - Data Science
- Разработка
 - Scala, Java, Python, C, Lisp
- IT Infrastructure
 - Администрирование
 - Сопровождение
 - Архитектура

Big Data проекты в банках, телекоме и в рознице.

Правила вебинара



Активно участвуем



Задаем вопрос в чат или голосом



Off-topic обсуждаем в Slack #канал группы или #general



Вопросы вижу в чате, могу ответить не сразу

Цели вебинара

1

Познакомимся с очередями сообщений

2

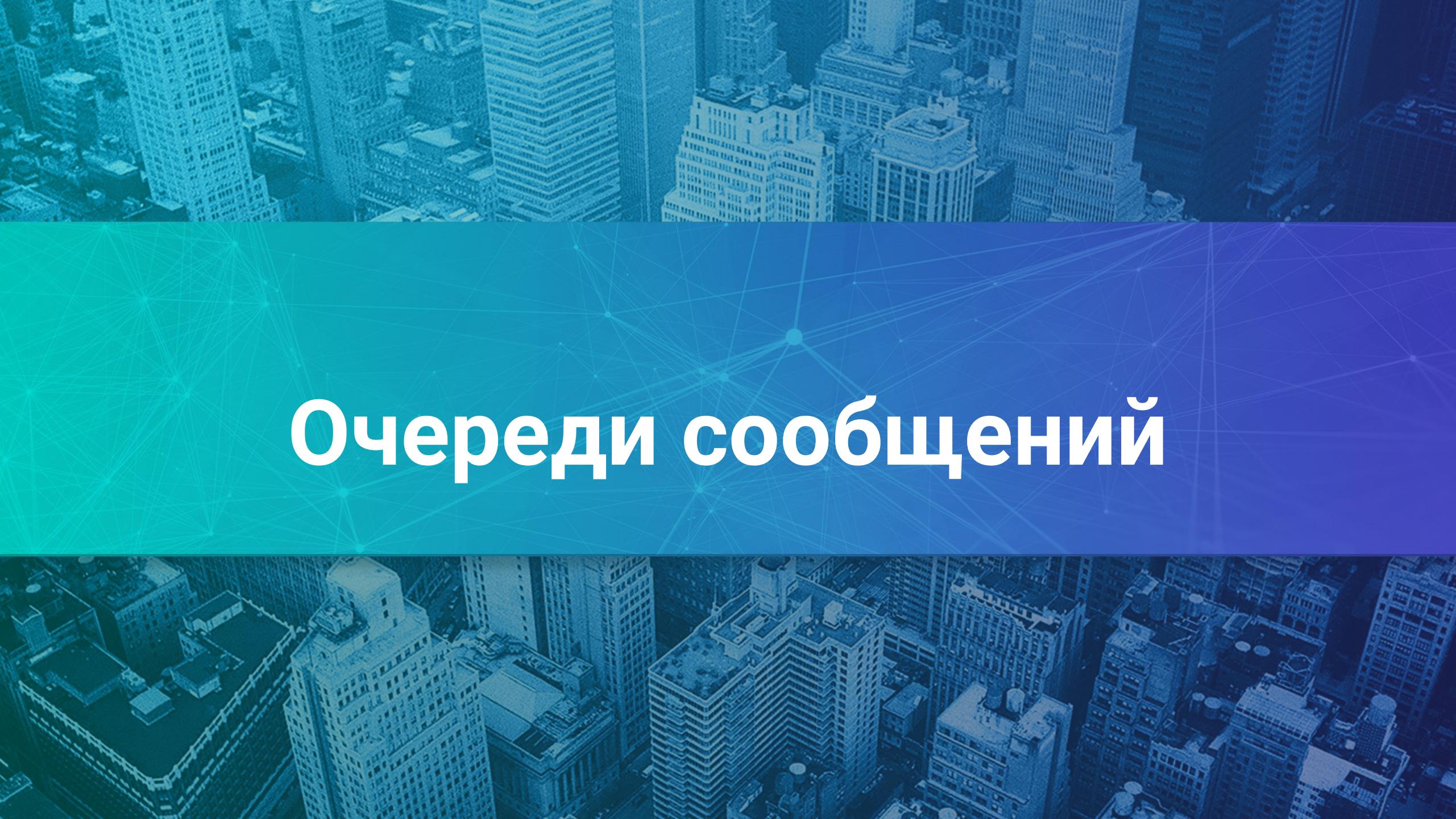
Познакомимся с RabbitMQ

3

Познакомимся с Kafka

Смысл

- 1 Узнаем как использовать очереди сообщений
- 2 Сможем начать работать с RabbitMQ
- 3 Сможем начать работать с Kafka

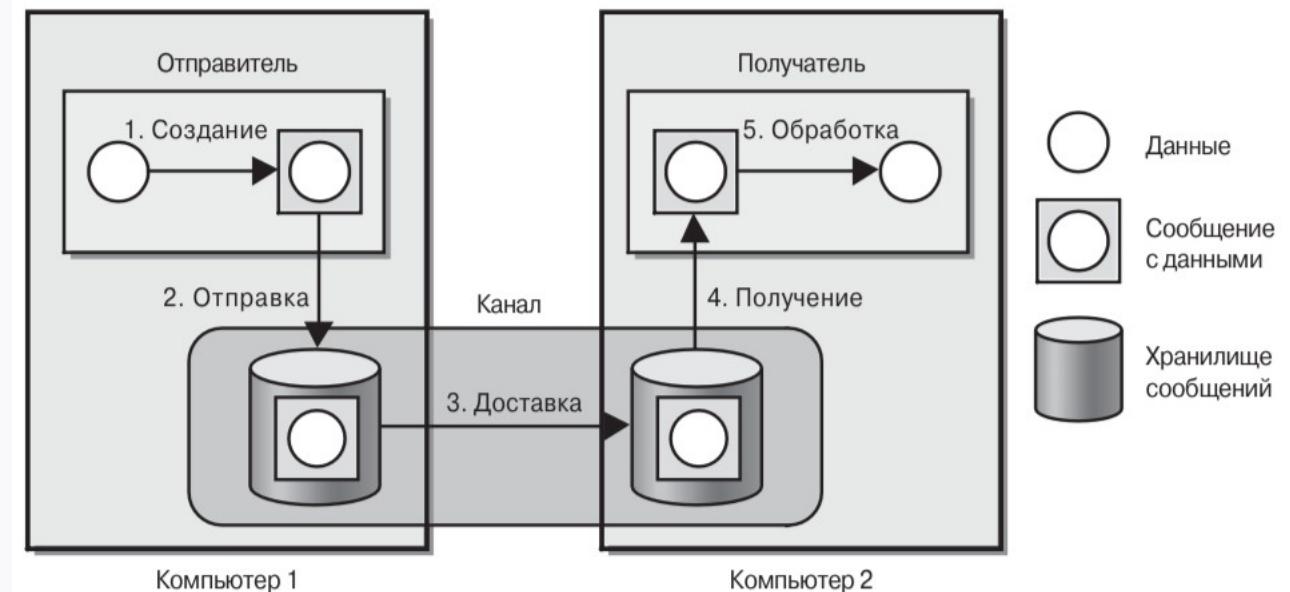


Очереди сообщений

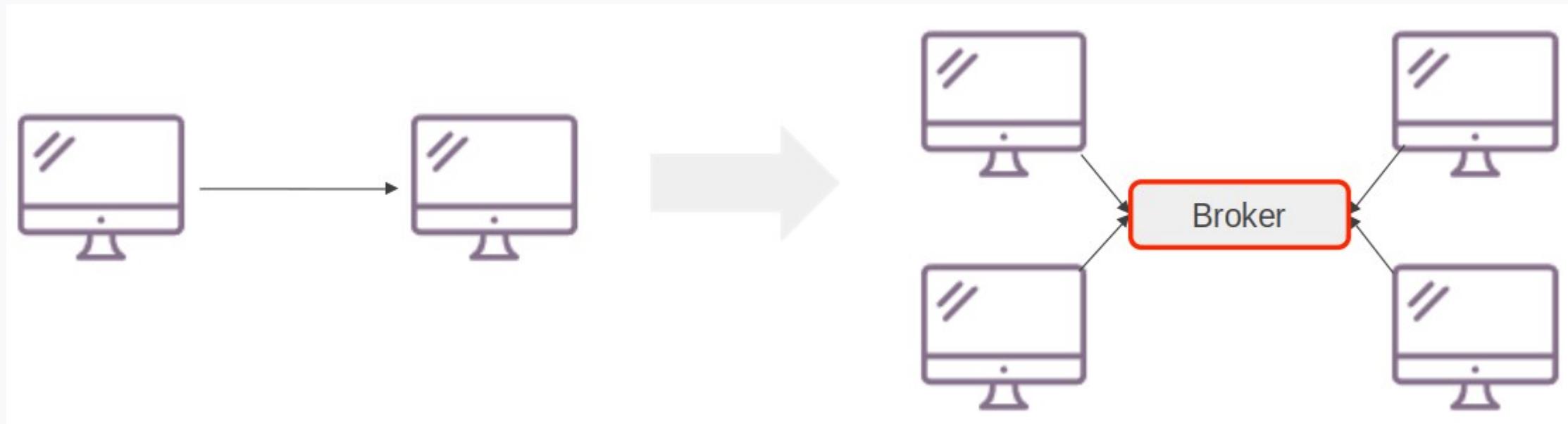
Система организации потоков

Система организации потоков – система нежёсткого реального времени, которая делает данные доступными, когда клиентское приложение хочет их видеть.

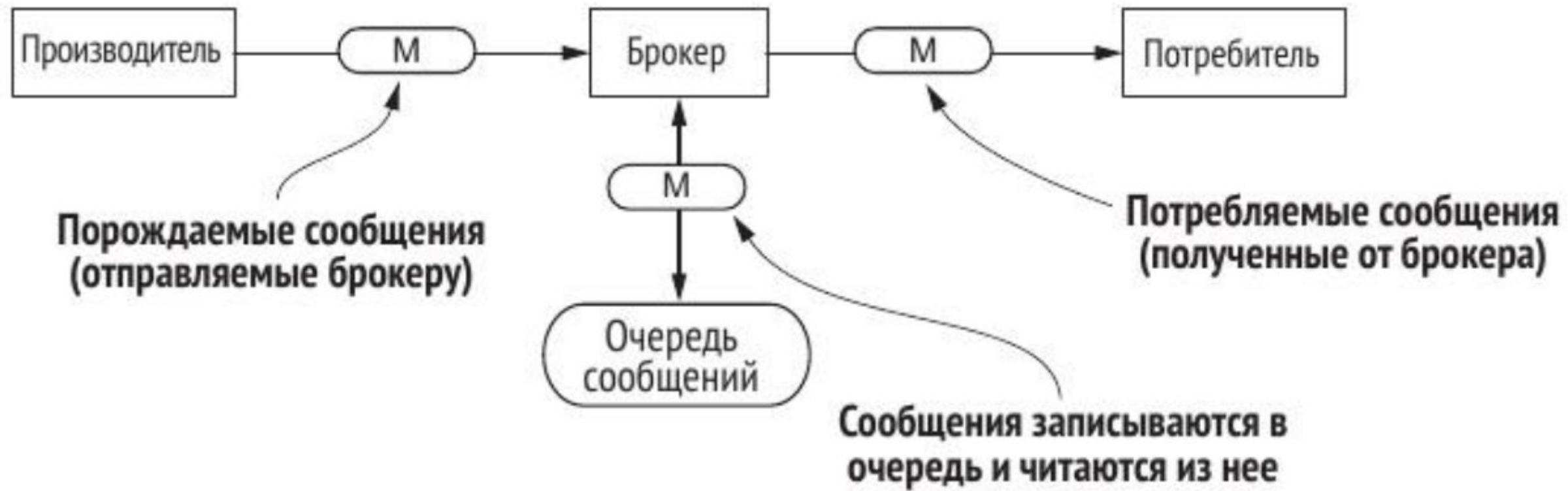
- Слабая связанность
- Асинхронное взаимодействие
- Рассогласование во времени
- Регулирование нагрузки
- Надёжное взаимодействие
- Удалённое взаимодействие



Обмен сообщениями

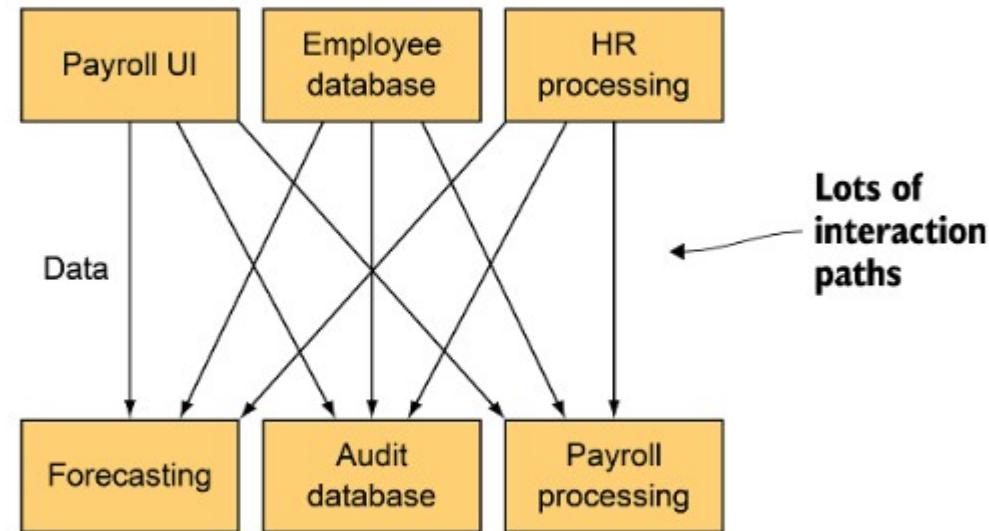


Производитель, брокер, потребитель



Пример традиционной архитектуры

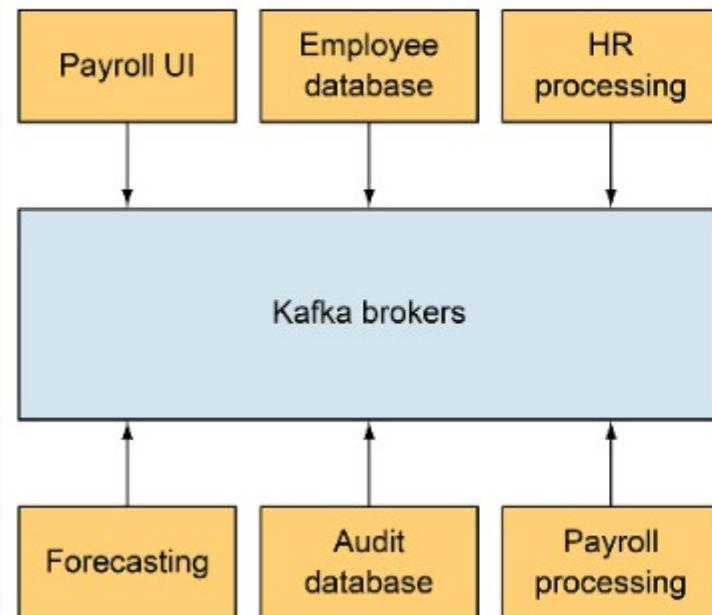
If you had a type of Richardson Maturity Model for data ingestion, this would be one of the lowest levels! These applications are sharing information with various APIs and are dependent on various uptime and interface issues.



This diagram only shows the applications on the bottom row needing data from the top row. Often this is bi-directional, but you can already see the sprawl of communication channels.

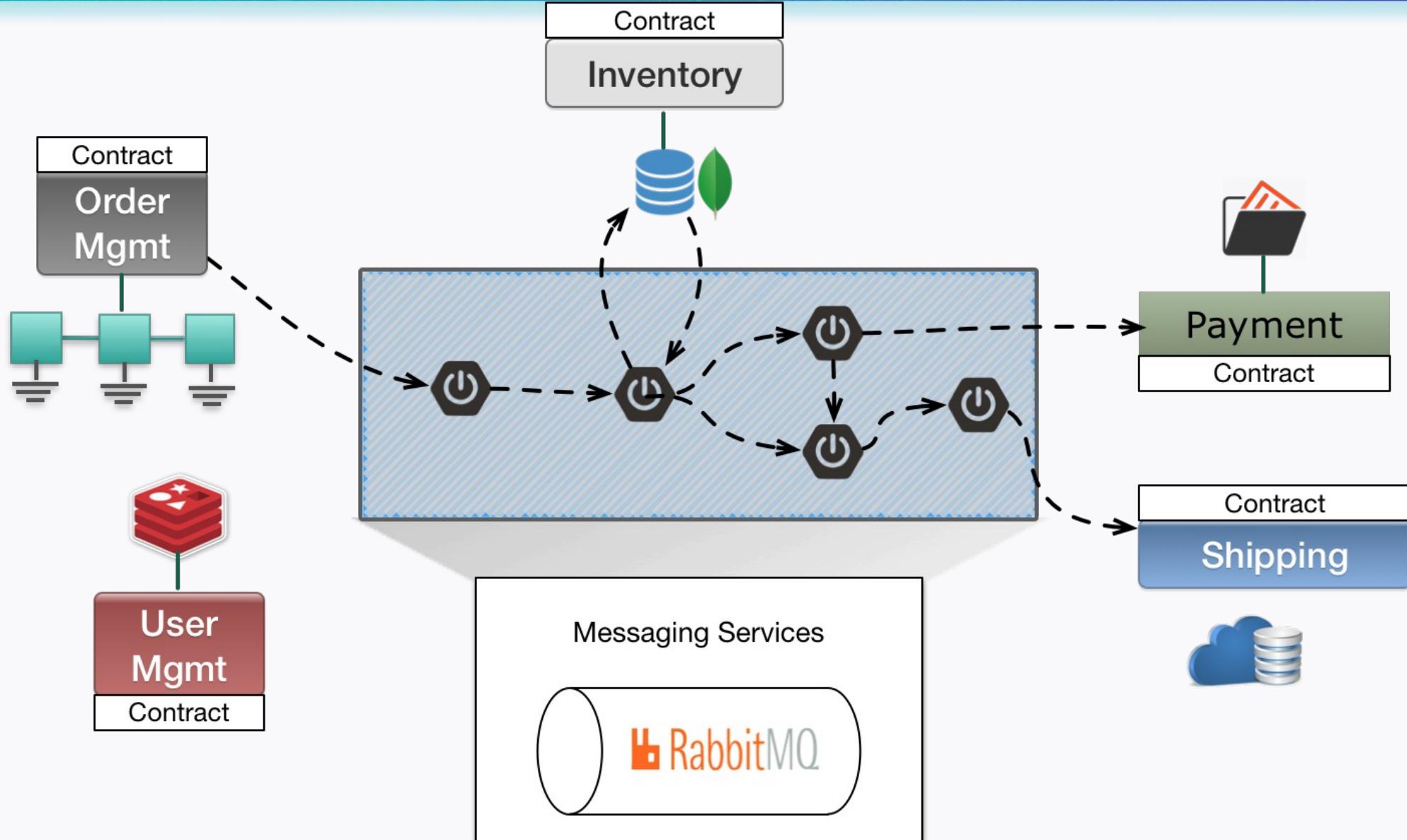
Пример архитектуры, ориентированной на события

Same data source and sinks. The interface between applications is now only Kafka.

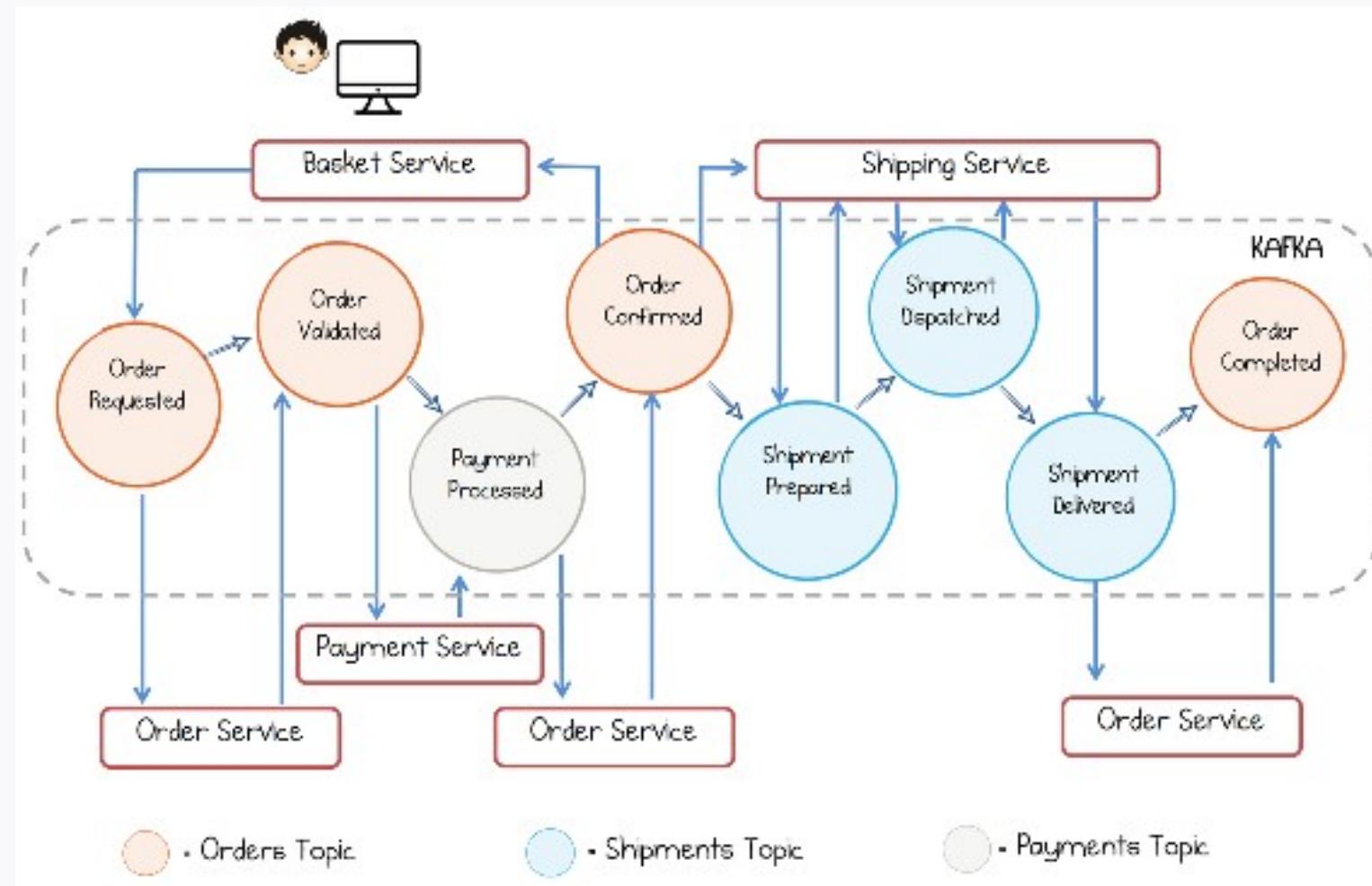


Besides application decoupling of interfaces, now the applications have the chance to produce and consume data at their own pace and not be tied to a specific application uptime or availability.

Пример архитектуры, ориентированной на события



Шаблон "Событийное сотрудничество"

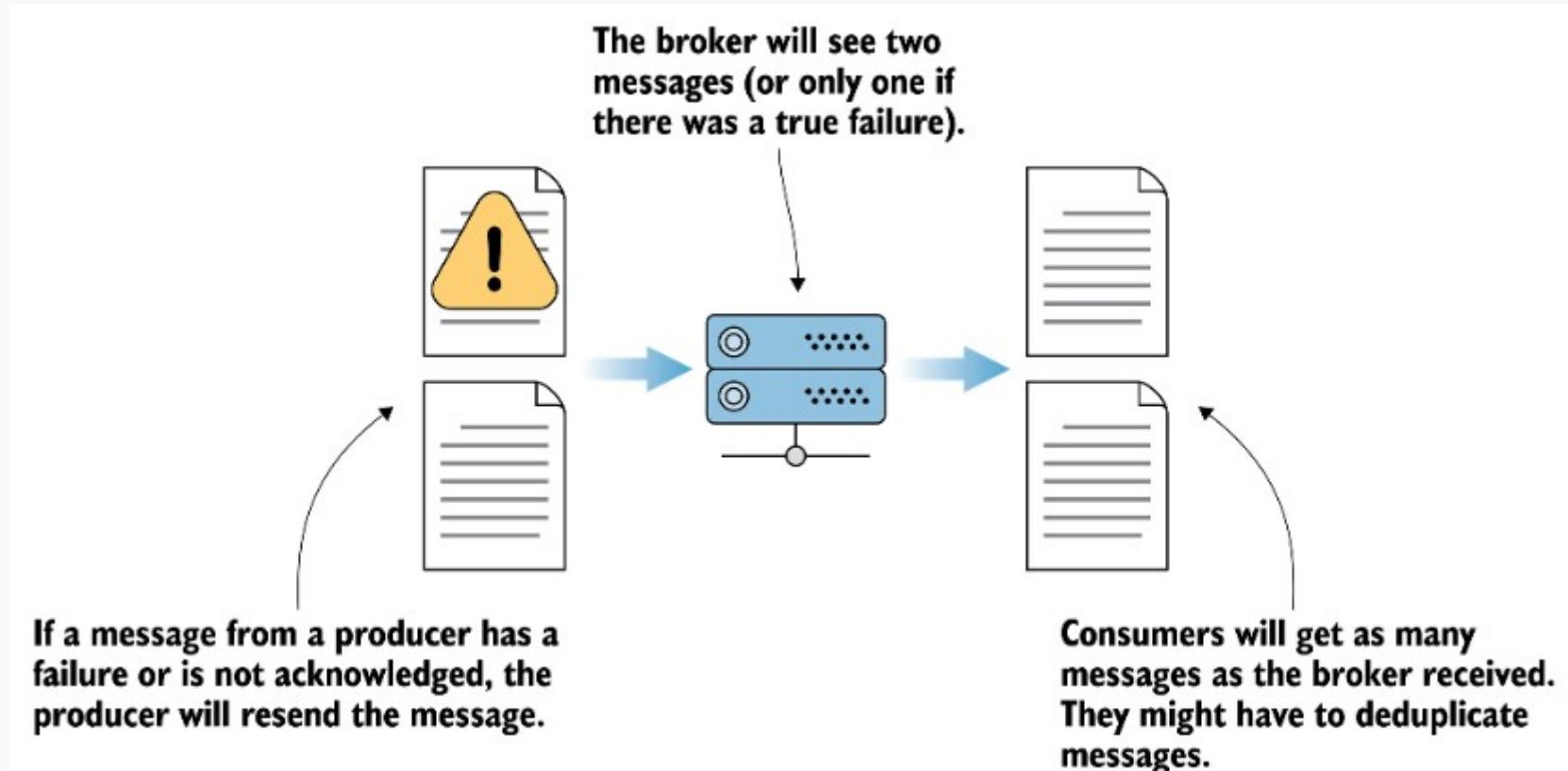


Семантика доставки

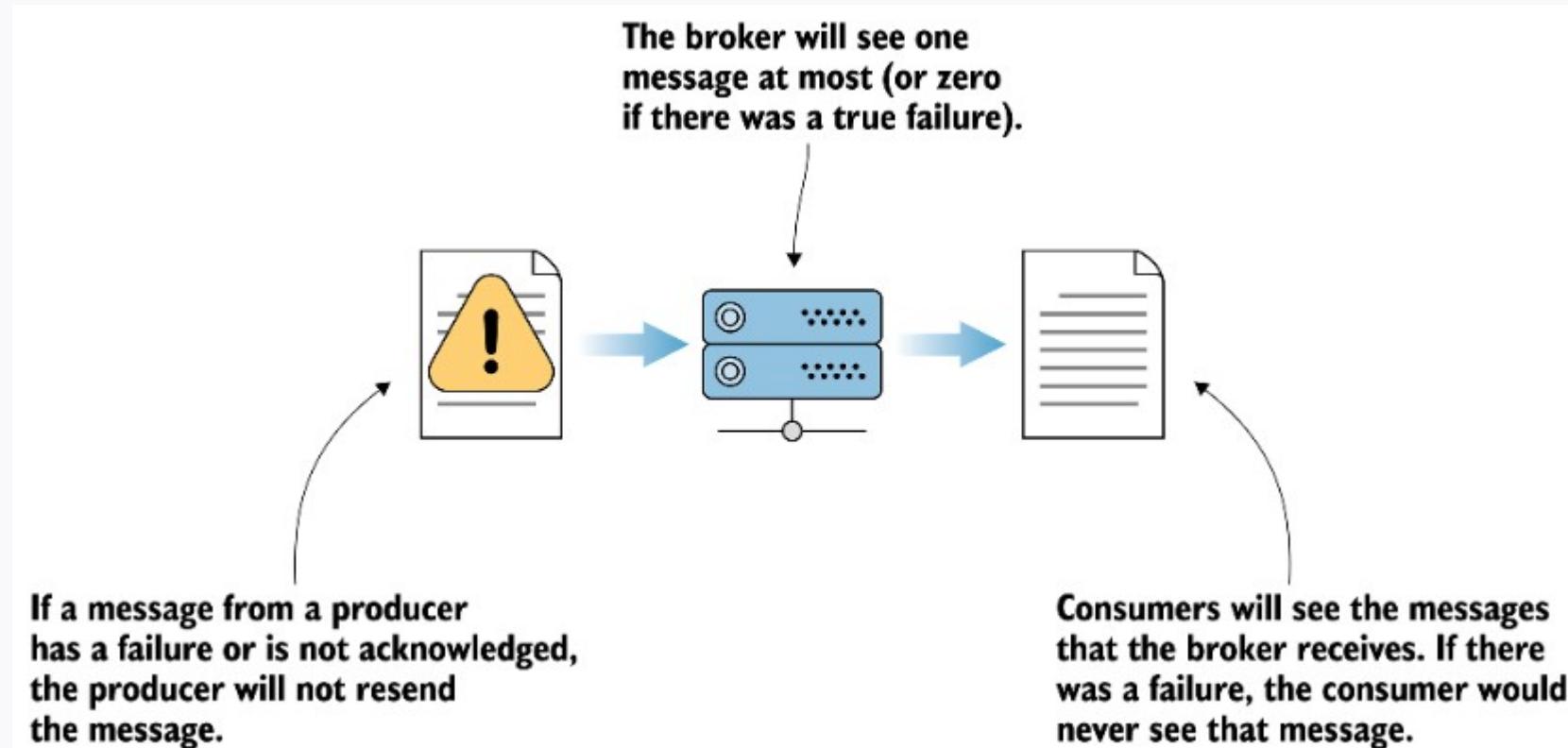
Семантика доставки

- **At-least-once** (*Не менее одного раза*) - сообщение не может потеряться, но может быть обработано несколько раз
- **At-most-once** (*Не более одного раза*) - сообщение может потеряться, но никогда не будет обработано дважды
- **Exactly-once** (*Ровно один раз*) - сообщение не может потеряться и обрабатывается ровно один раз

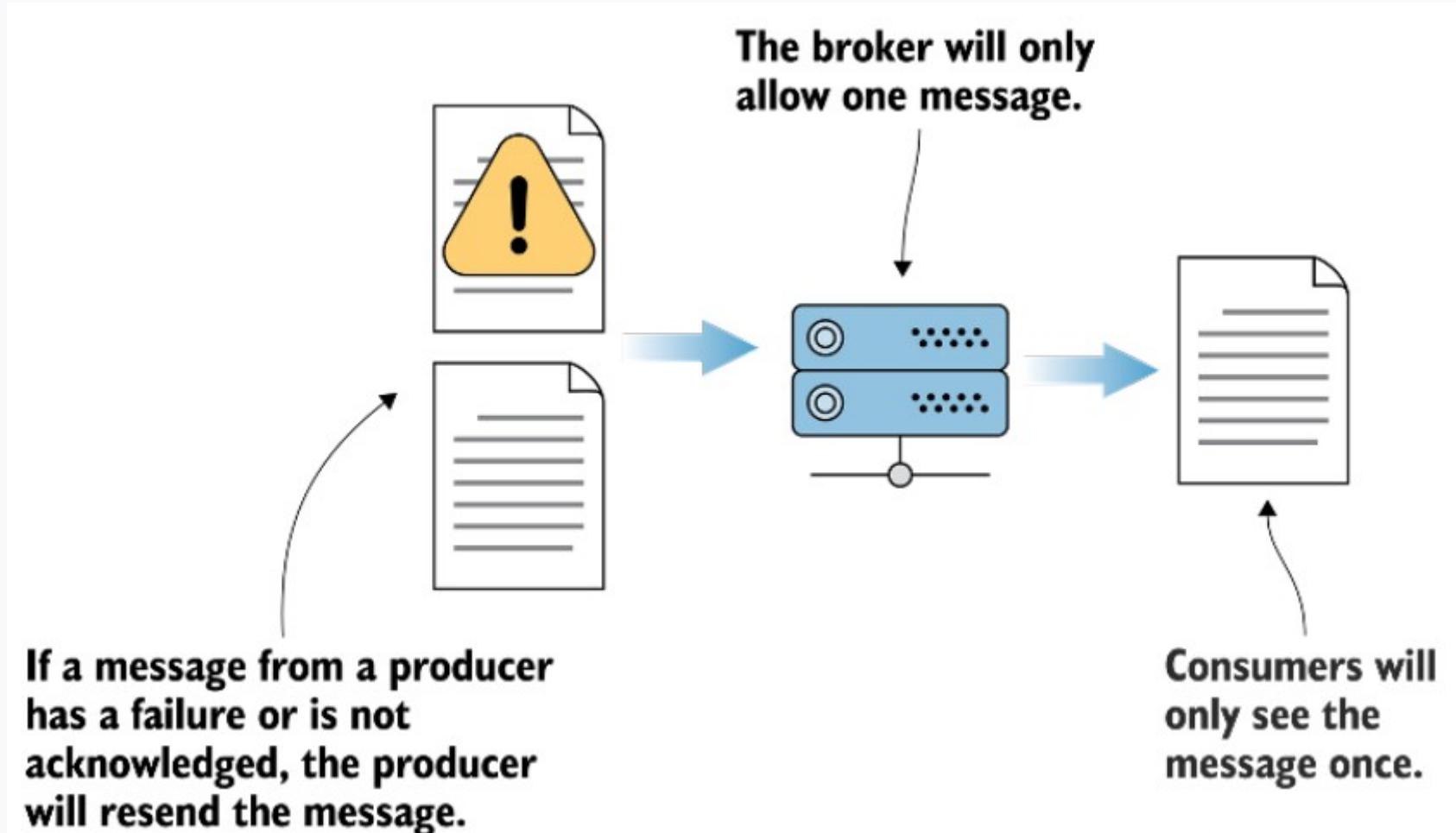
At-least-once



At-most-once



Exactly-once



The background features a high-angle aerial photograph of a dense urban area with numerous skyscrapers. Overlaid on this background is a network visualization consisting of a central white dot and a web of thin, light-colored lines extending towards the edges of the frame.

RabbitMQ

RabbitMQ



RabbitMQ — брокер сообщений общего назначения, разработанный по модели "Умный Брокер / Пассивный Клиент"

- Открытый код
- Открытые стандартные протоколы
 - AMQP 0.9.1 (complex routing), 1.0
 - JMS, MQTT, STOMP, HTTP
- Подключаемая аутентификация (например, LDAP, OAUTH2)
- Большое количество клиентов и плагинов: Java, Python, .Net и т.д.
- Multi-tenant isolation

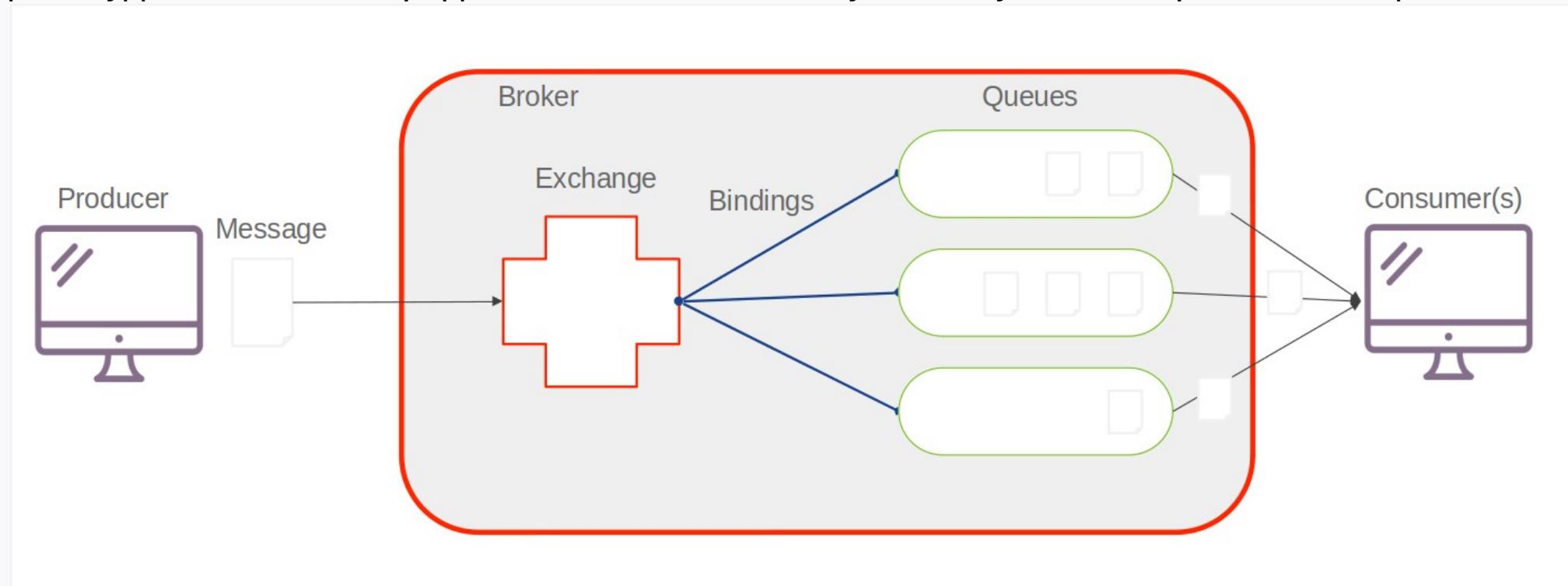
Tanzu RabbitMQ

VMware Tanzu RabbitMQ — коммерческий вариант RabbitMQ

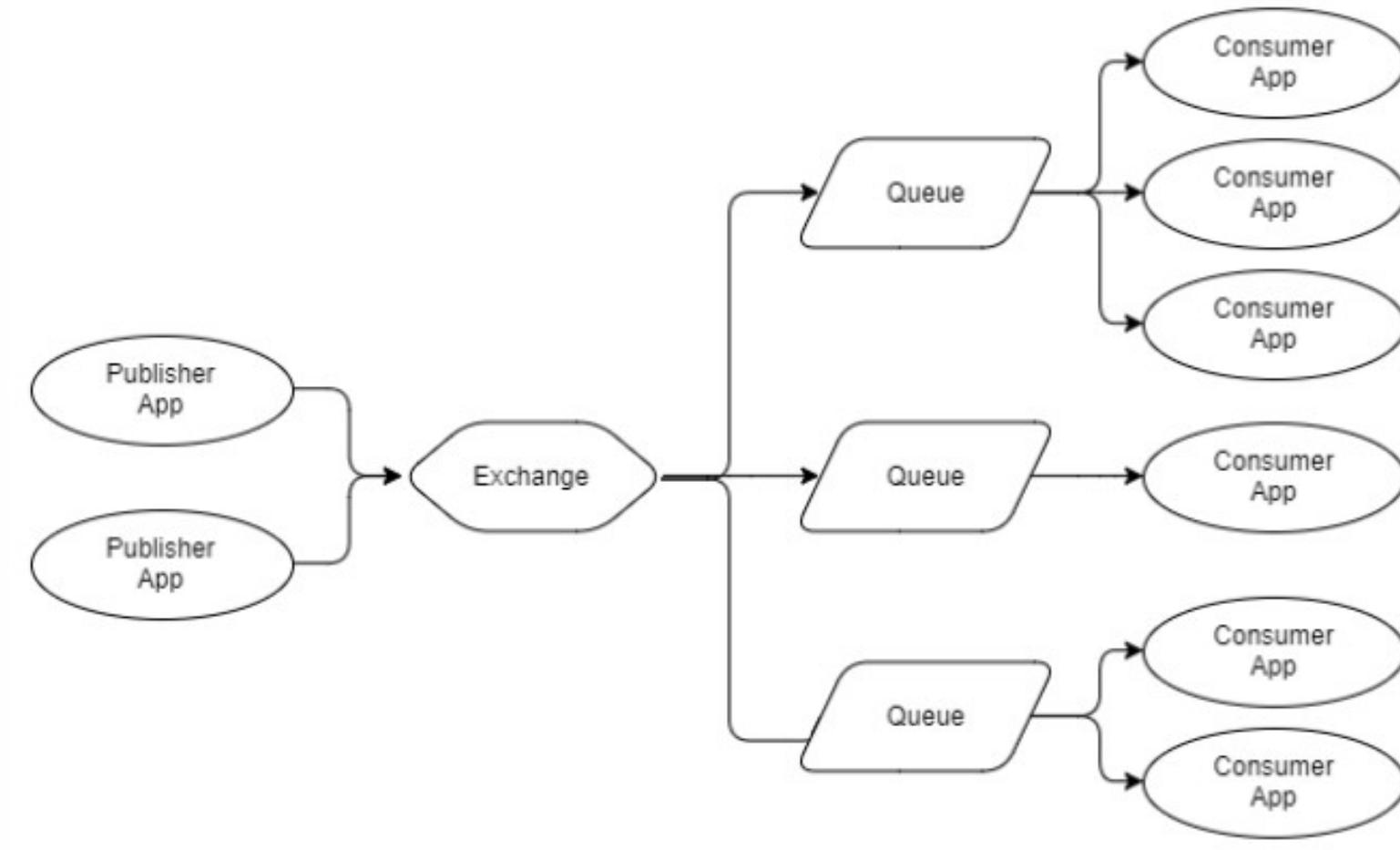
- Репликация схем между сайтами
- Компрессия данных при передачи между узлами
- Безопасный по умолчанию
- Прямая обратная связь с инженерами
- Расширенный Жизненный цикл Поддержки
- Поддержка 24 x 7

Работа RabbitMQ

- Отправитель шлет сообщение в Exchange
- RabbitMQ шлет подтверждение отправителю
- Exchange пересыпает сообщения в очереди и другие Exchange
- RabbitMQ отправляет сообщение всем получателям
- Получатели отправляют подтверждения
- Сообщение удаляется из очередей, как только все получатели успешно прочли сообщение



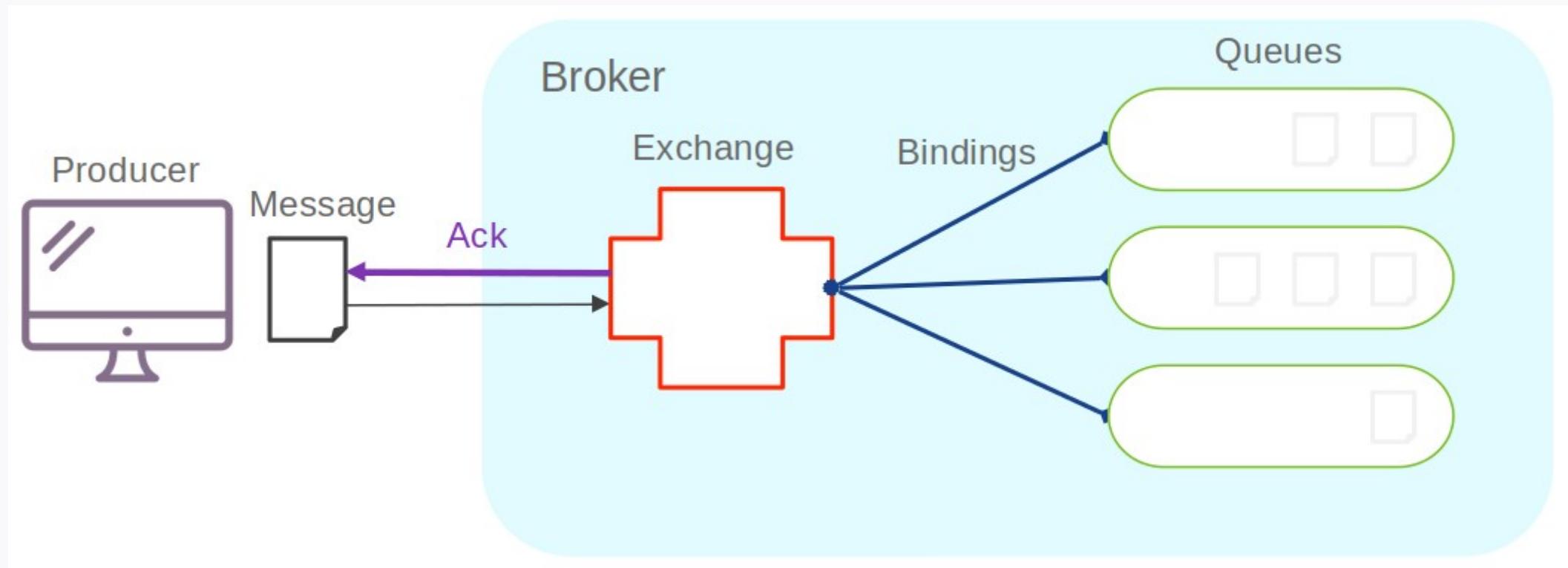
Отправители и Получатели



Publisher Confirms

Механизм для производителей подтверждения успешной обработки:

- сообщение отбрасывается (например, не направляется в очередь)
- сообщение было добавлено в соответствующие очереди или записано на диск



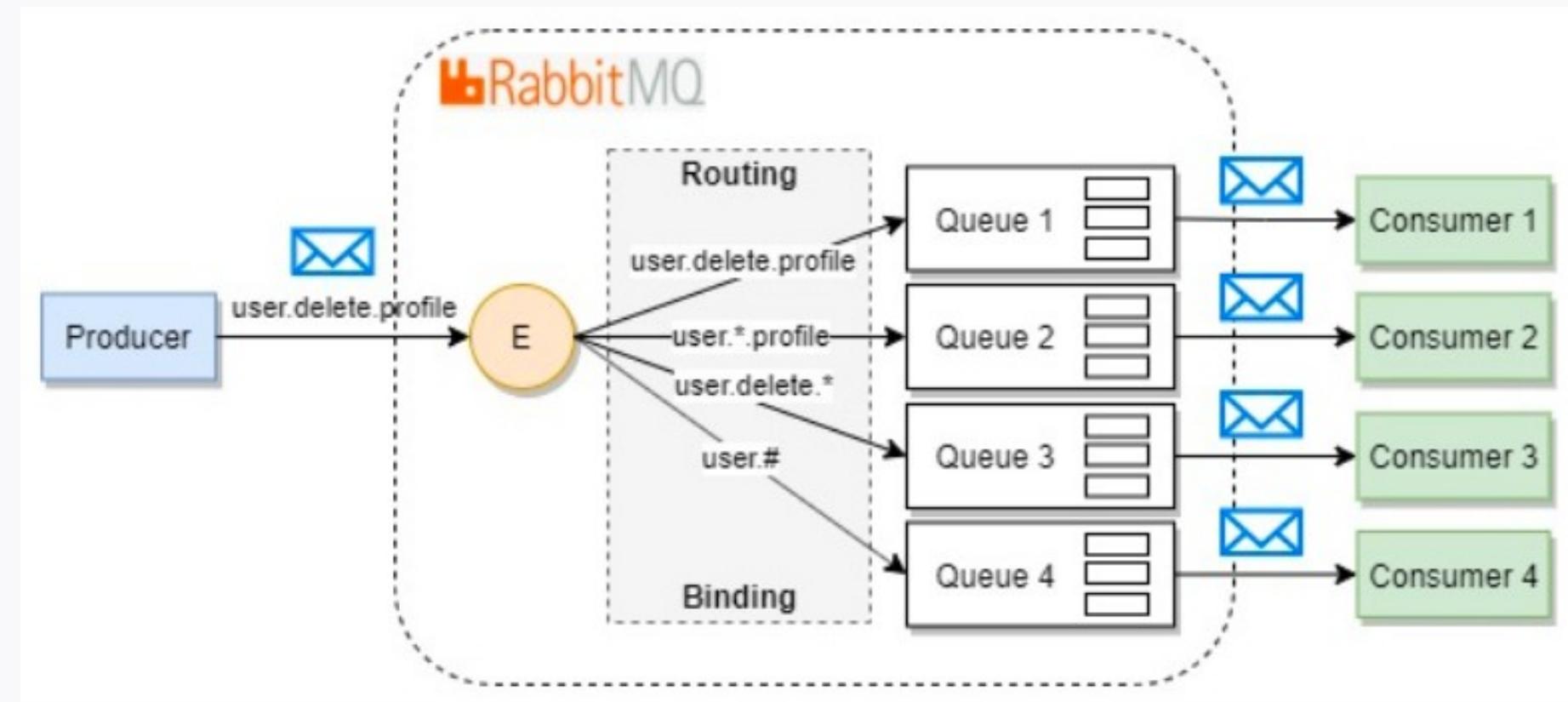
Consumer Acknowledgement

	No-Ack	Consume without acknowledge
Consume	Ack	Acknowledge message
	Multi-Ack	Acknowledge multiple messages at once
Reject	Reject	Reject individual, delivered message
	Nack	Reject messages in bulk

Маршрутизация сообщений

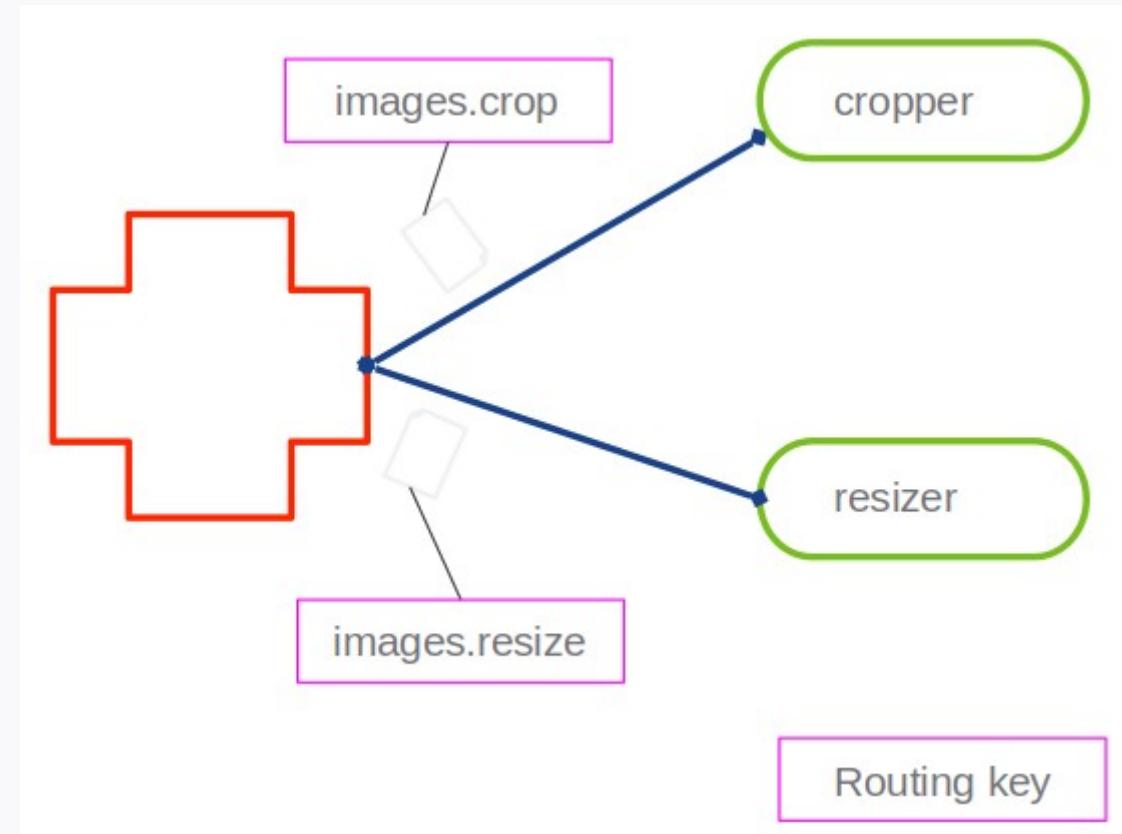
Exchange - это инструкции по маршрутизации для распространения сообщений

Binding - это правила, определяющие, какие очереди относятся к сообщению



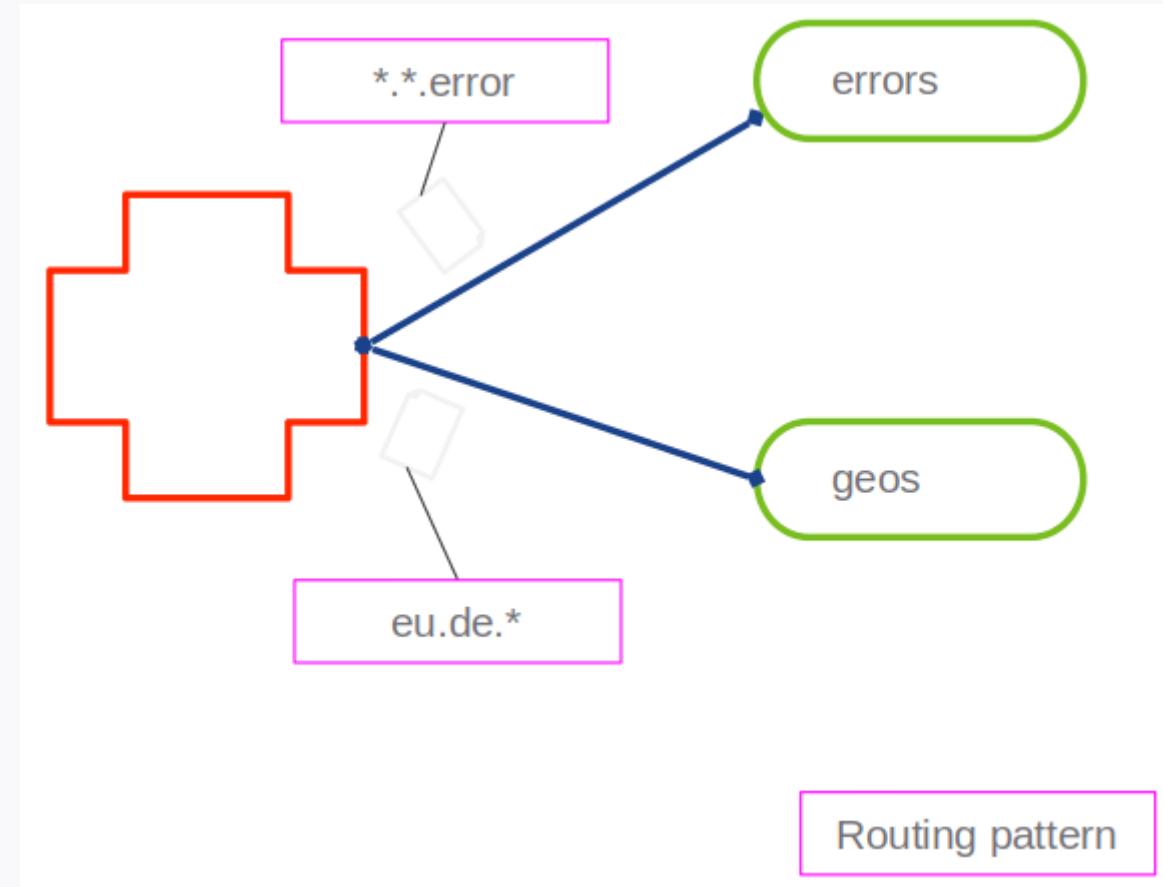
Direct Exchange

Direct Exchange доставляет сообщения в очереди, когда ключ маршрутизации сообщений точно совпадает с ключом привязки очереди



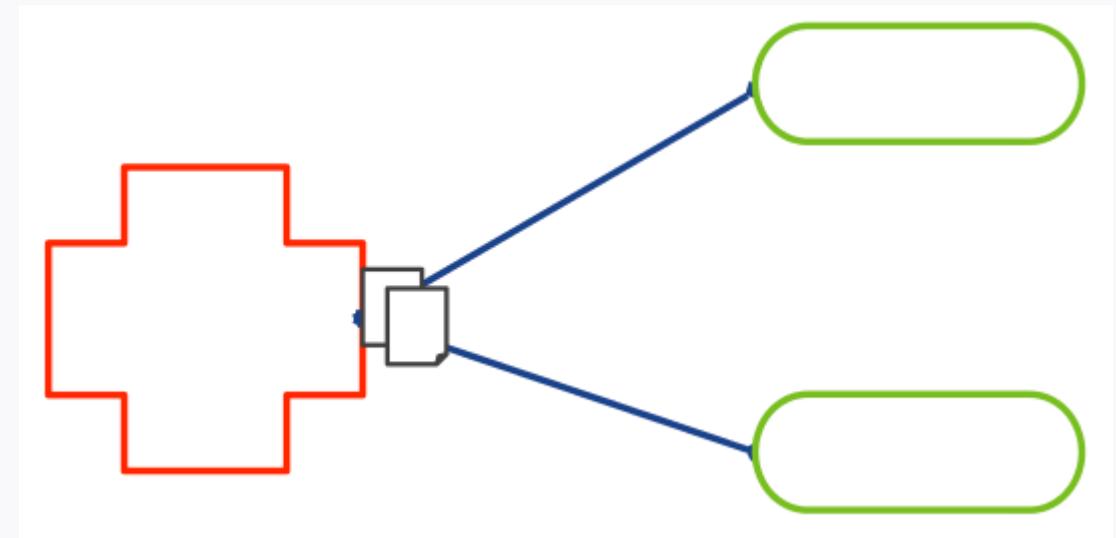
Topic Exchange

Topic Exchange доставляет сообщения в очереди, когда подстановочный знак (wildcard) совпадает между ключом маршрутизации и ключом привязки очереди



Fan-out Exchange

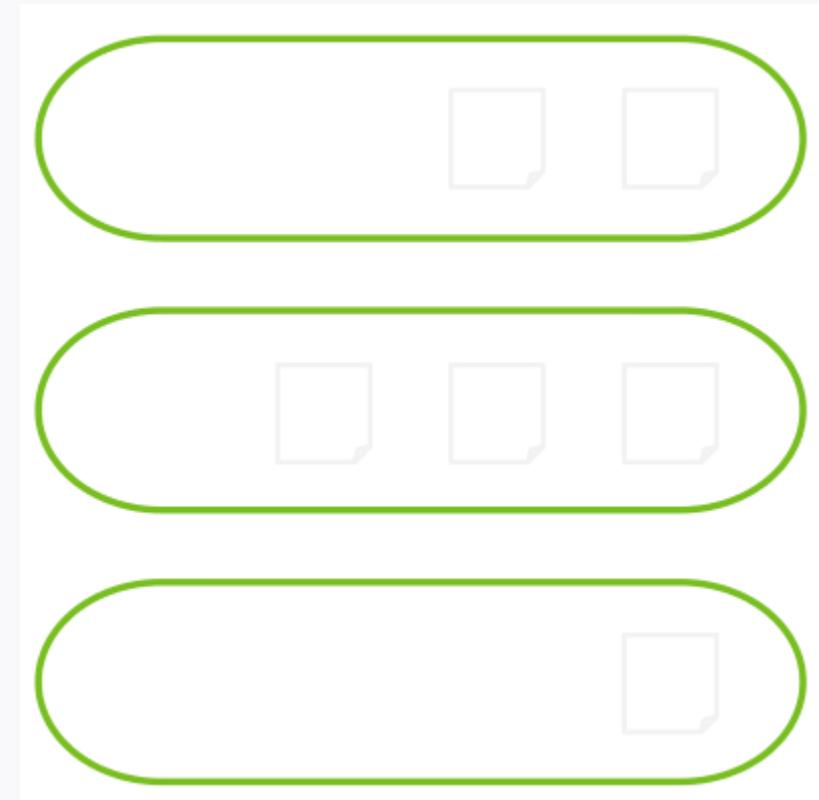
Fan-out Exchange доставляет сообщения во все очереди независимо от ключей маршрутизации или соответствия шаблону



Queues

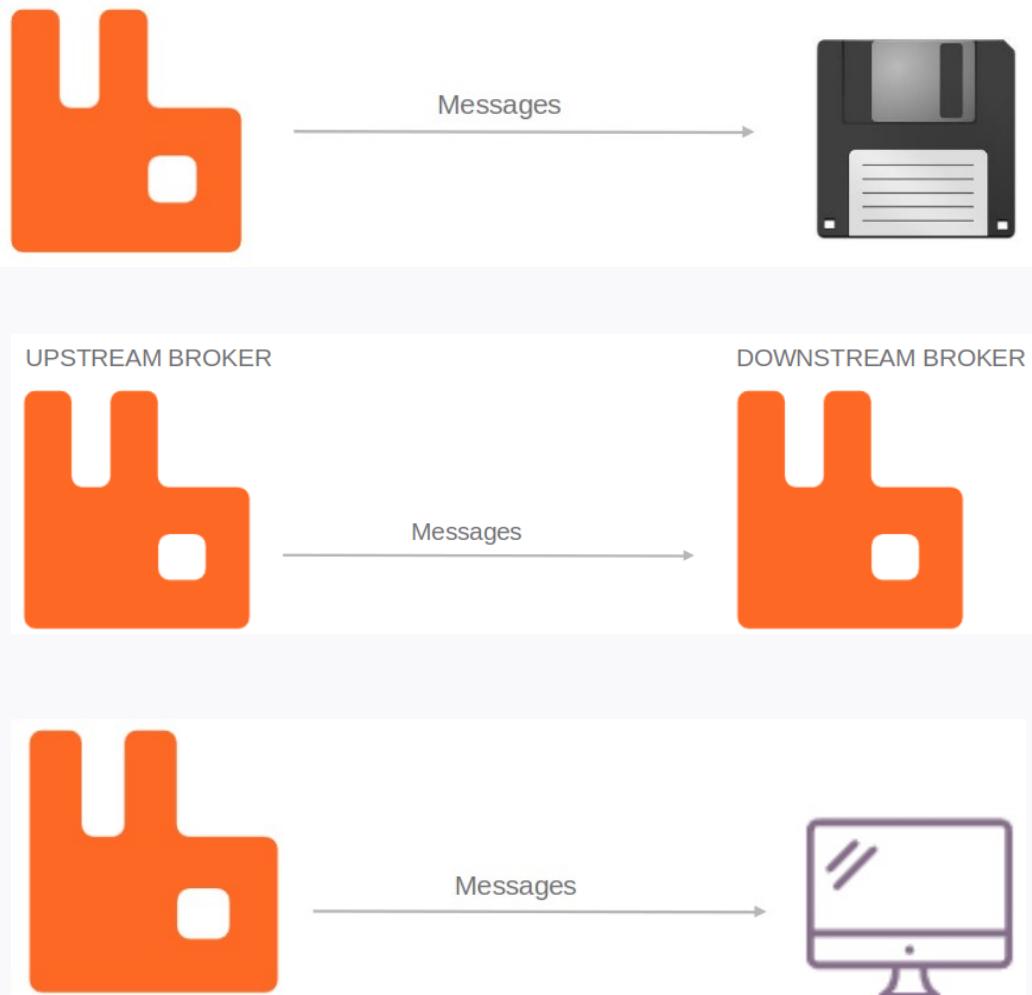
Очередь - это буфер, в котором хранятся сообщения

Очереди могут быть настроены так, чтобы иметь **набор поведений** (сохраняемость, срок действия по времени, максимальная длина и т.д.)



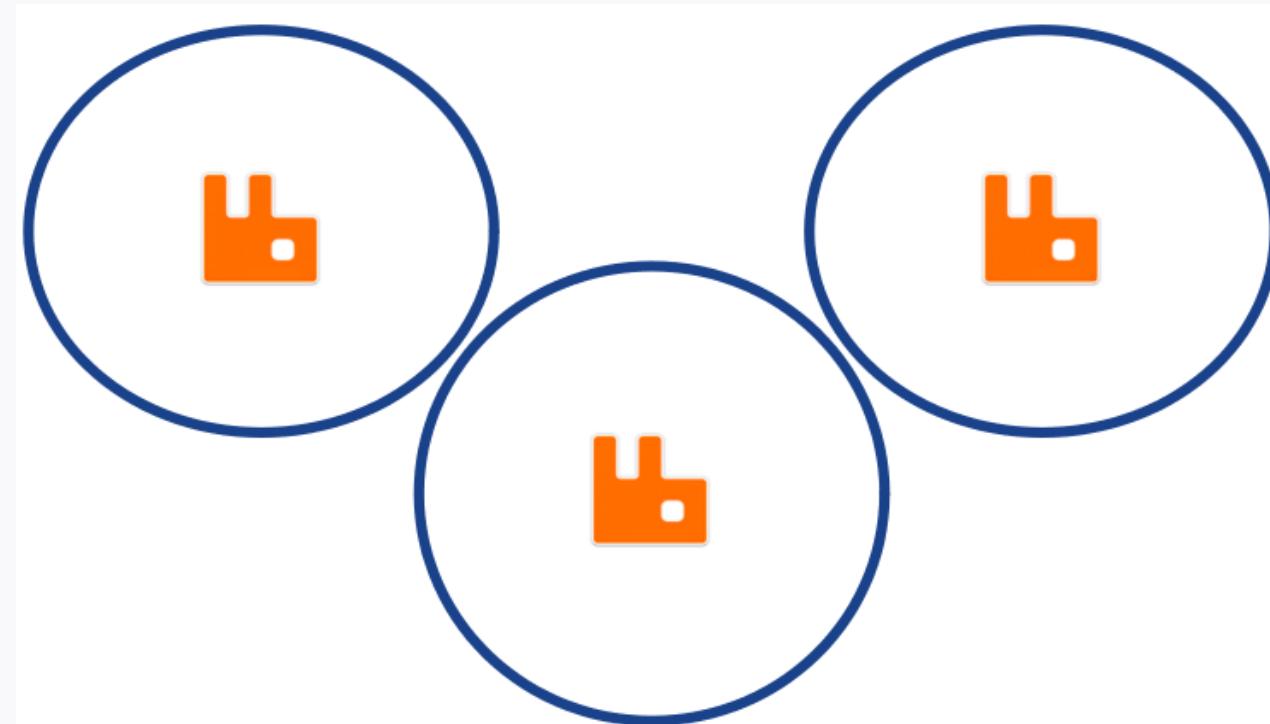
Путь сообщений

- Сообщения могут быть настроены на сохранение на диске для обеспечения отказоустойчивости в случае сбоя сервера
- Сообщение может быть реплицировано или перемещено на другой брокер RabbitMQ (например, для распределения нагрузки) с помощью подключаемого модуля *shovel/federation* или кластеризации
- Сообщение доставляется потребителям, которые подключены к брокеру RabbitMQ и подписаны на очереди



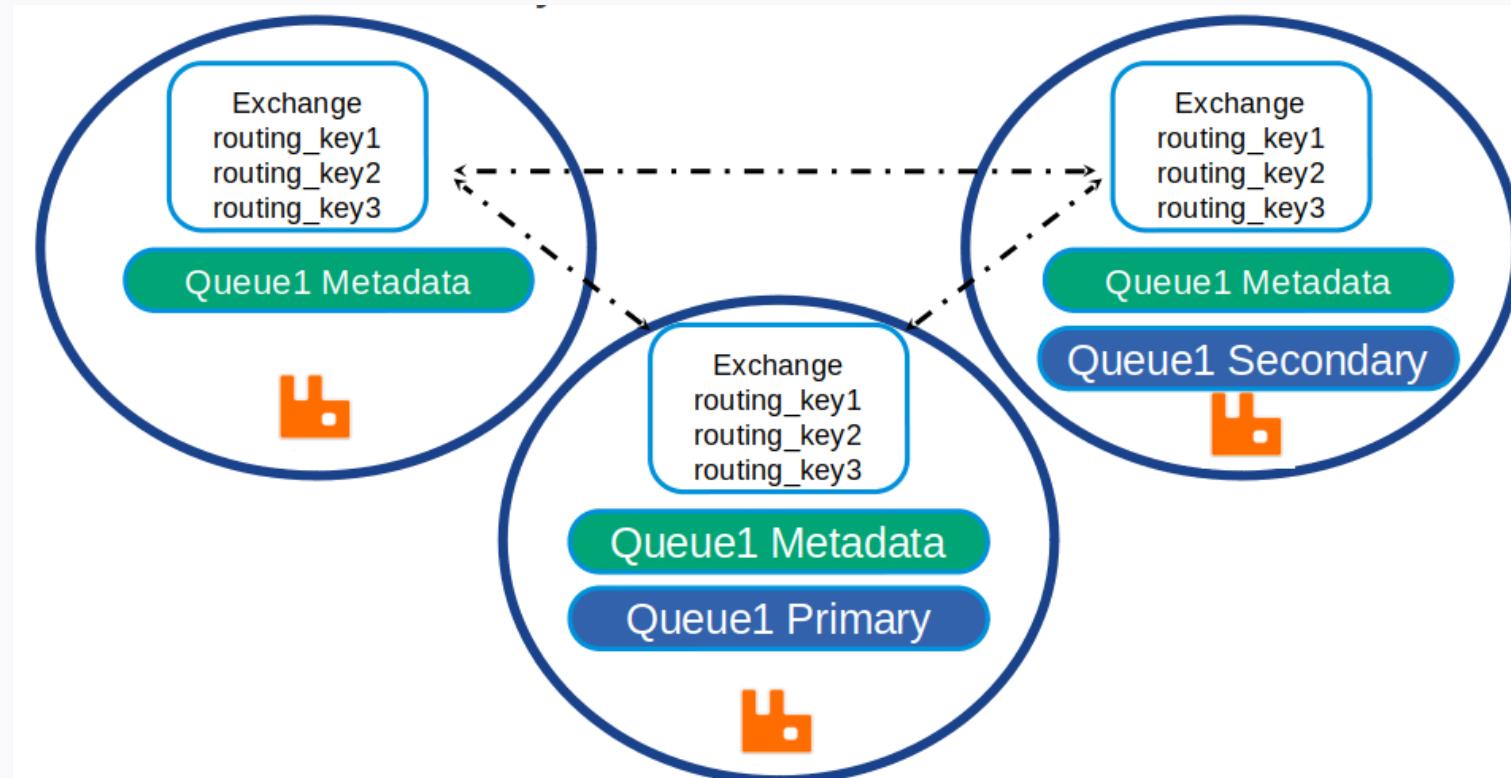
Кластеризация

- Кластеризация обеспечивает **высокую доступность и повышенную пропускную способность** за счет формирования единого концептуального брокера из нескольких машин в одном месте (критична задержка в сети)
- Оптимизирована для обеспечения согласованности (*consistency*) и устойчивости к разделению (*partition tolerance*) в теореме CAP



Кластеризация - Архитектура

- Все метаданные, необходимые для брокера, реплицируются на все узлы
- Нереплицированные очереди сообщений находятся на одном брокере
- Реплицированные очереди имеют первичную и вторичную реплики, размещенные на разных брокерах
- Очереди доступны с любого узла



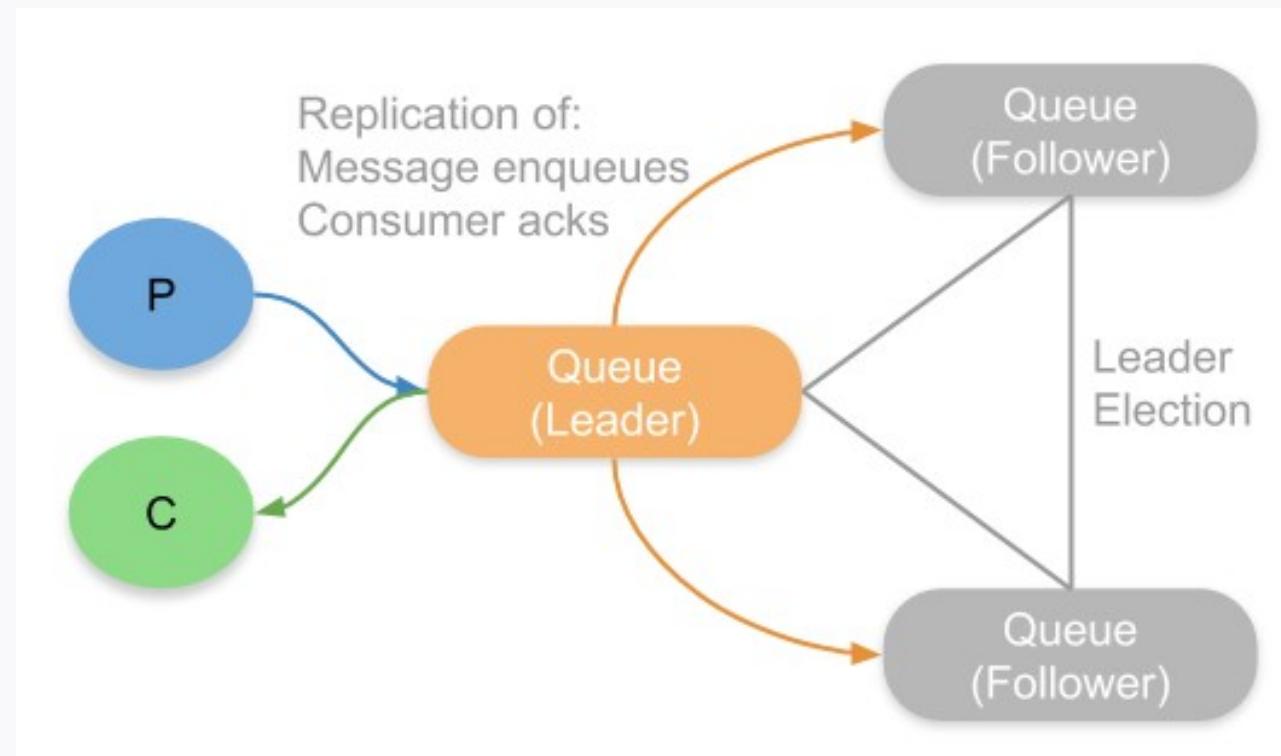
Стратегии обработки разделения

Три способа автоматической работы с сетевыми разделениями:

- **pause-minority** - автоматически приостанавливайте работу узлов кластера, которые считают себя в меньшинстве (т.е. меньше или равны половине от общего числа узлов) после того, как другие узлы выйдут из строя - выбирает согласованность, а не доступность
- **pause-if-all-down** - автоматически приостанавливает работу узлов кластера, которые не могут достичь ни одного из узлов
- **autoheal** - автоматически выберет раздел-победитель, если считается, что разделение произошло, и перезапустит все узлы, которые не входят в раздел-победитель - оптимизирован для обеспечения доступности

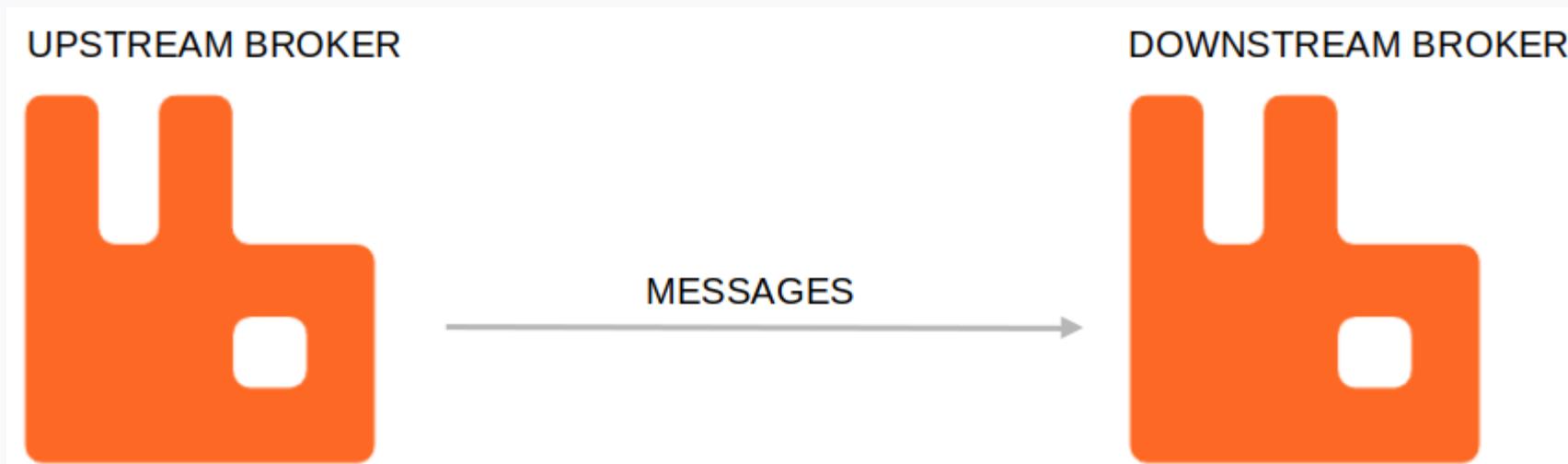
Quorum Queues

- **Quorum Queues** - это современный тип очереди для RabbitMQ, реализующий устойчивую реплицируемую очередь FIFO на основе алгоритма консенсуса Raft
- Обеспечивают более высокую пропускную способность, чем зеркальные очереди, с более быстрым и плавным отказоустойчивым переключением



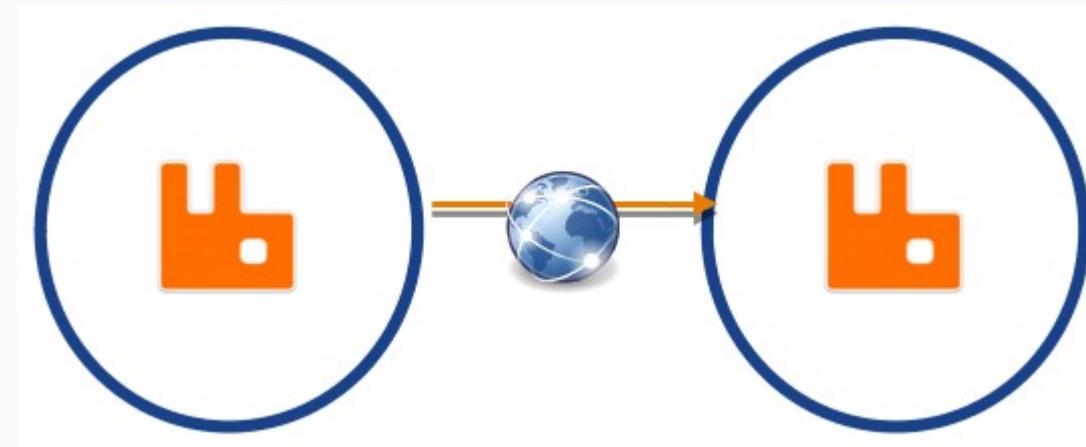
Federation

- Федерация обеспечивает целенаправленное распределение Exchanges и очередей путем репликации или перемещения сообщений между слабо связанными брокерами/клUSTERами
- Оптимизирована для доступности (*availability*) и устойчивости к разделению (*partition tolerance*) в теореме CAP
- Federation plugin потребляет сообщения с вышестоящего узла и внутренне повторно публикует их на своем брокере



Shoveling

- Shoveling **перемещает сообщения** между брокерами / кластерами для балансировки нагрузки в концептуально аналогичной форме федерации
- Оптимизирован для доступности (*availability*) и устойчивости к разделению (*partition tolerance*) в теореме CAP



Shovel vs Federation

Shovel	Federation
Потребляет сообщения из исходной очереди и отправляет их в Exchange на узле назначения	Подключает брокер к одному или нескольким вышестоящим брокерам для распространения сообщений через федеративные очереди / Exchange
Может быть настроен на источнике, пункте назначения или на стороннем брокере	Более высокоуровневый
Сценарий использования: <ul style="list-style-type: none">Легко перемещает сообщения из очереди в Exchange того же или другого брокера	Сценарии использования: <ul style="list-style-type: none">Если потребители переносятся с одного узла на другой, федеративная очередь позволит это сделать без пропуска сообщений или их повторной обработкиПоддерживает одну и ту же "логическую" очередь, распределенную по многим брокерам

Distributed RabbitMQ - Сравнение

	Federation/ Shovel	Clustering
Теорема CAP	Доступность + Устойчивости к разделению	Согласованность + Устойчивости к разделению
Брокер	Полностью отдельные	Единый концептуальный брокер
Версии RMQ/Erlang	Могут быть разными	Должны быть одинаковыми
Коммуникации	Через ненадежные каналы WAN. Связь осуществляется через AMQP, требуются пользователи / разрешения	Через надежные каналы LAN. Связь осуществляется обменом сообщениями между узлами Erlang, требуется общий файл cookie Erlang
Соединения	Брокеры подключены одно или двусторонним способом	Все узлы соединяются со всеми другими узлами
Особенности	Можно указать федеративные Exchanges	Все или ничего для кластеризации
Видимость подключенных клиентов	Можно видеть только очереди в этом брокере	Можно видеть очереди на всех узлах

Практика

- Запускаем
 - docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:management
- Скачиваем Management CLI
 - wget http://localhost:15672/cli/rabbitmqadmin
 - chmod +x rabbitmqadmin
- Management GUI: http://localhost:15672 (guest/guest)
- Создаём очередь с именем test
 - rabbitmqadmin declare queue name=test
 - rabbitmqadmin list queues vhost name node messages
- Публикуем сообщение
 - rabbitmqadmin publish exchange=amq.default routing_key=test payload="hello, world"
- Читаем сообщение
 - rabbitmqadmin get queue=test ackmode=ack_requeue_false

Администрирование

- **rabbitmqctl** - общее управление
- **rabbitmq-diagnostics** - для диагностики
- **rabbitmq-plugins** - для управления плагинами
- **rabbitmqadmin** - для управления по HTTP API



Kafka

Что такое Kafka?

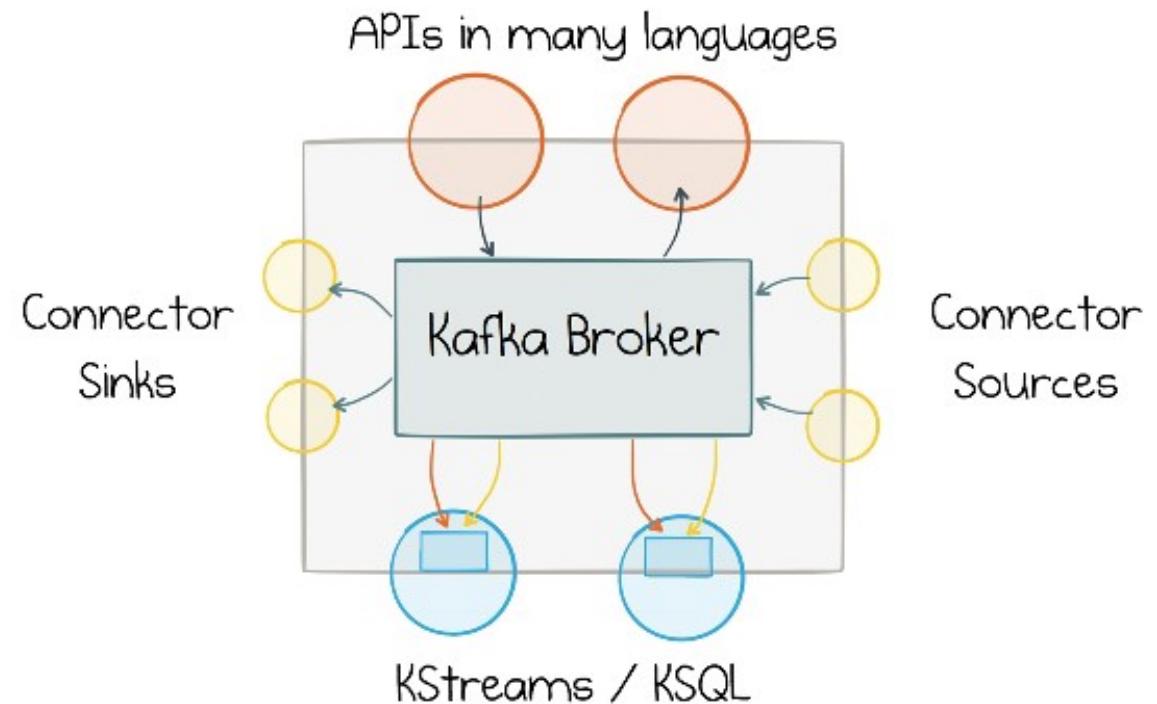
Kafka – это распределенная потоковая платформа, которая обладает тремя основными возможностями:

- публиковать и подписываться на записи (очереди сообщений)
- хранить записи с отказоустойчивостью
- обрабатывать потоки по мере их возникновения

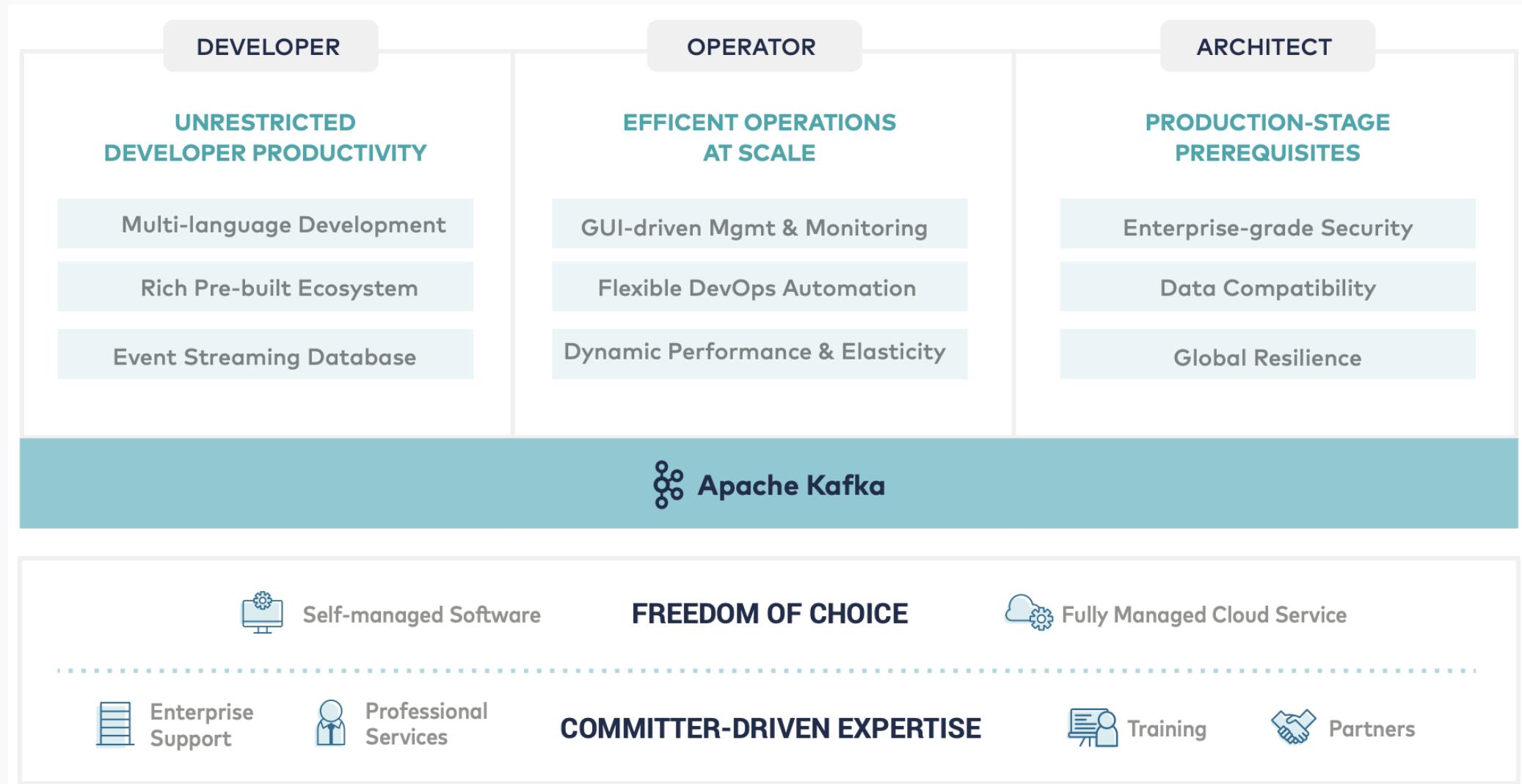
Kafka – Платформа потоковой обработки данных

- Брокер сообщений
- Producer API – отправка сообщений
- Consumer API – чтение сообщений
- Connect API – импорт / экспорт данных
- Streams API – обработка потоков

Kafka: a Streaming Platform

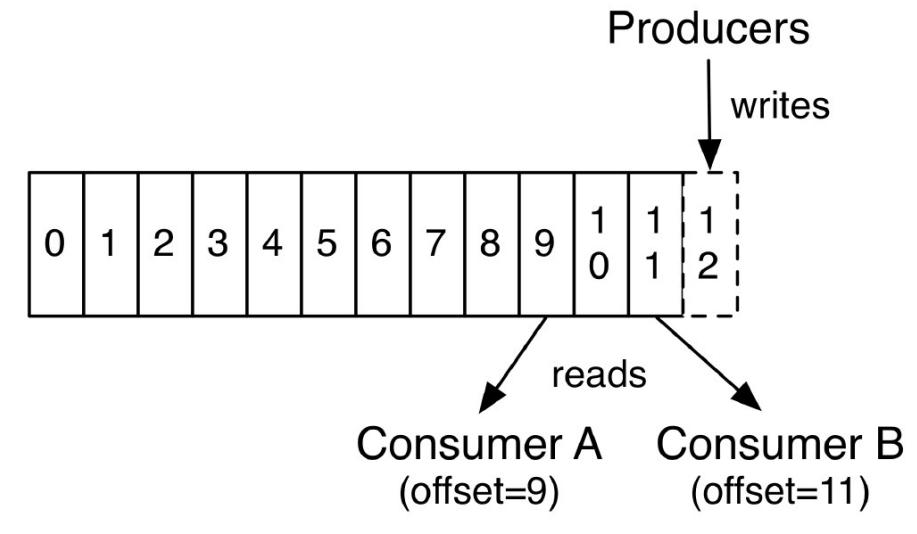


Confluent Platform

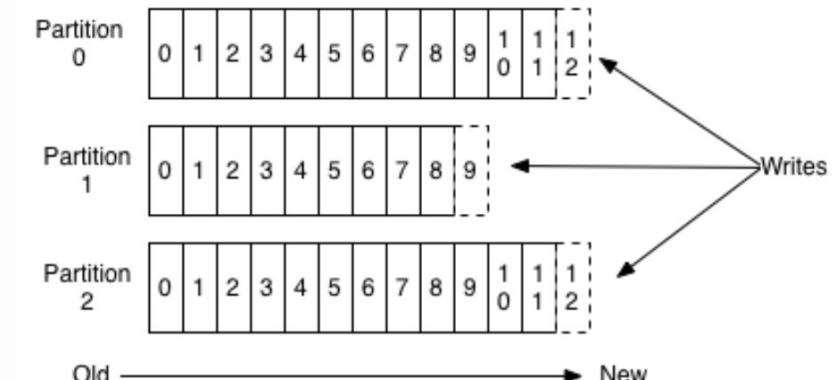


Основные концепции

- **Record** – элемент данных ключ-значение
- **Topic** – имя потока с данными
- **Producer** – процесс, публикующий записи
- **Consumer** – процесс, читающий записи
- **Offset** – позиция записи
- **Partition** – единица параллелизма темы



Anatomy of a Topic



Запускаем Kafka

1) Скачиваем и разворачиваем

```
$ tar -xzf kafka_2.13-3.0.0.tgz  
$ cd kafka_2.13-3.0.0
```

2) Запускаем сервисы

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties  
$ bin/kafka-server-start.sh config/server.properties
```

3) Создаём тему

```
$ bin/kafka-topics.sh --create --topic quickstart --bootstrap-server localhost:9092
```

4) Запишем что-нибудь в тему

```
$ bin/kafka-console-producer.sh --topic quickstart --bootstrap-server localhost:9092  
This is my first event  
This is my second event
```

5) Прочитаем записи

```
$ bin/kafka-console-consumer.sh --topic quickstart --from-beginning --bootstrap-server localhost:9092  
This is my first event  
This is my second event
```

Kafka в Docker

```
version: '2'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    hostname: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

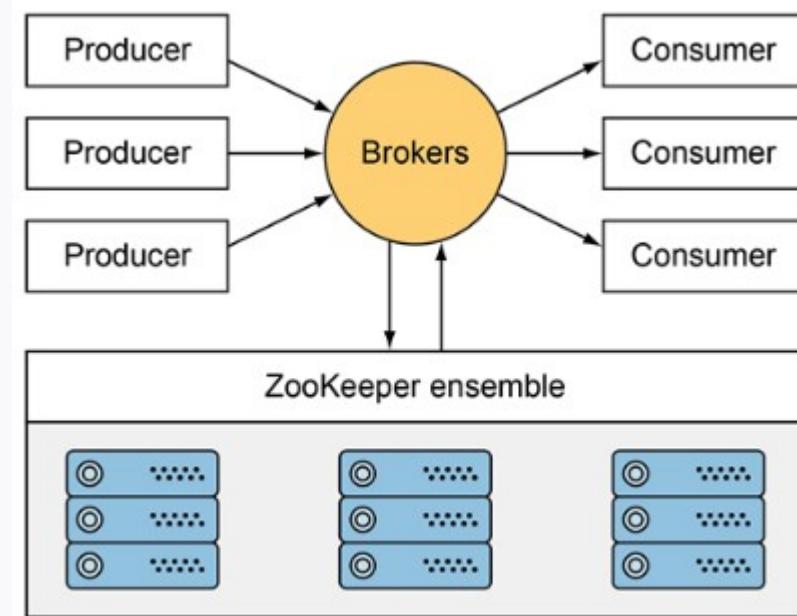
  broker:
    image: confluentinc/cp-kafka:latest
    hostname: broker
    depends_on:
      - zookeeper
    ports:
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

Основные операции

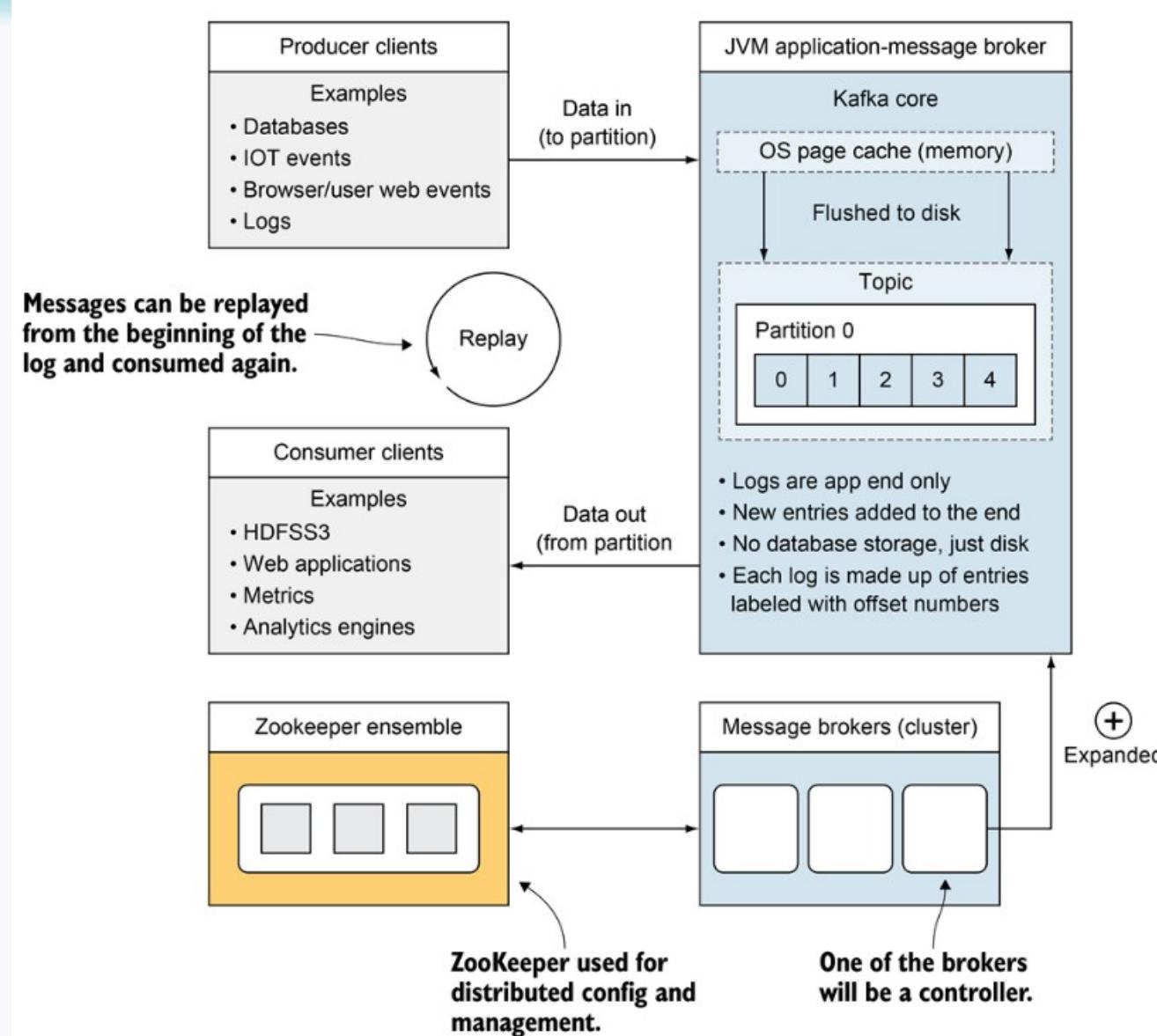
- `zookeeper-server-start.sh` – запуск Zookeeper
- `zookeeper-server-stop.sh` – останов Zookeeper
- `kafka-server-start.sh` – запуск Kafka брокера
- `kafka-server-stop.sh` – останов Kafka брокера
- `kafka-console-producer.sh` – консольный Producer
- `kafka-console-consumer.sh` – консольный Consumer
- `kafka-topics.sh` – работа с темами (создать, удалить, изменить, посмотреть)
- `kafka-consumer-groups.sh` – управление группами consumer

Архитектура кластера Kafka

- **Broker** – управление данными, взаимодействие с клиентами
- **Zookeeper** – членство брокеров в кластере, выборы контроллера
- **Контроллер** – это брокер, который также отвечает за выбор ведущих реплик для секций



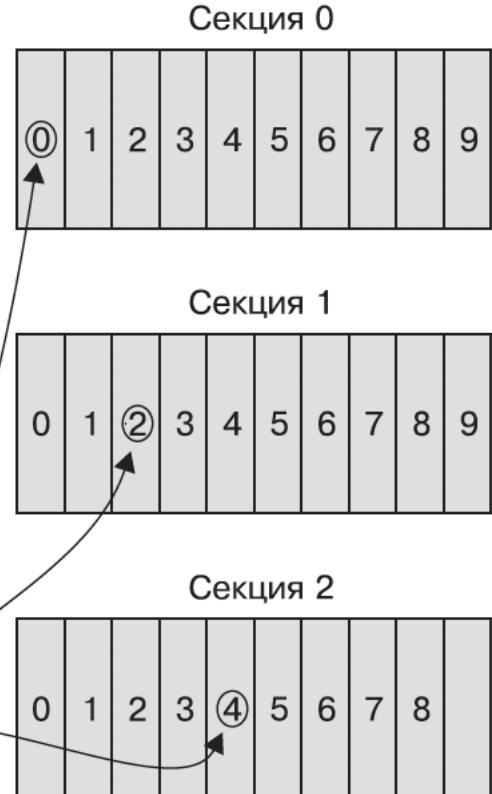
Работа кластера Kafka



Секции (Partitions, Разделы)

Топик с секциями

Данные поступают в один топик, но размещаются в отдельных секциях (0, 1 или 2). Поскольку ключей в этих сообщениях нет, они распределяются по секциям циклическим образом



Показанные в прямоугольниках
числа представляют собой
смещения сообщений

При поступлении сообщений
они записываются в назначаемые
генератором секции и добавляются
в конец журнала в соответствии
с временем поступления

Сообщения внутри всех
секций располагаются
в строго возрастающем
порядке, но никакой
упорядоченности сообщений
между различными секциями нет

Секции группируют данные по ключу

Входящие сообщения:

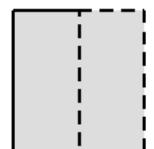
```
{foo, данные сообщения}  
{bar, данные сообщения}
```

Секция, в которую будет отправлено сообщение, определяется его ключами. Эти ключи не пусты.

Байтовое представление ключа используется для вычисления хеша

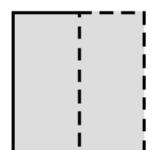
Секция 0

$\text{hashCode}(\text{fooBytes}) \% 2 = 0$



Секция 1

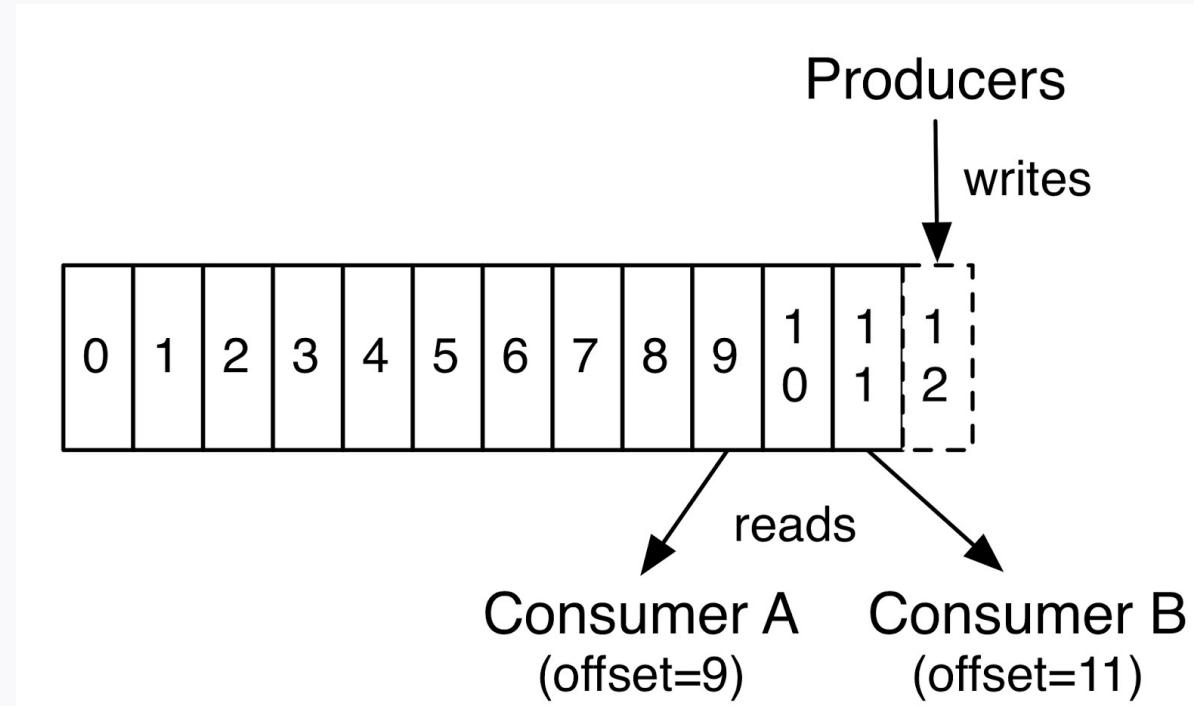
После определения секции сообщение добавляется в конец соответствующего журнала



$\text{hashCode}(\text{barBytes}) \% 2 = 1$

Commit Log

- Сообщения хранятся в виде бесконечной ленты
- Потребители подключаются для чтения сообщений
- Прогресс отслеживается через смещения (ссылки на ленту)



Функционирование журналов

- `log.dir` – место хранения журналов
- Подкаталог – тема
- Кол-во подкаталогов – кол-во секций
- Формат названия: имя–секции_номер-секции

/logs

/logs/topicA_0

В topicA — одна секция

/logs/topicB_0

В topicB — три секции

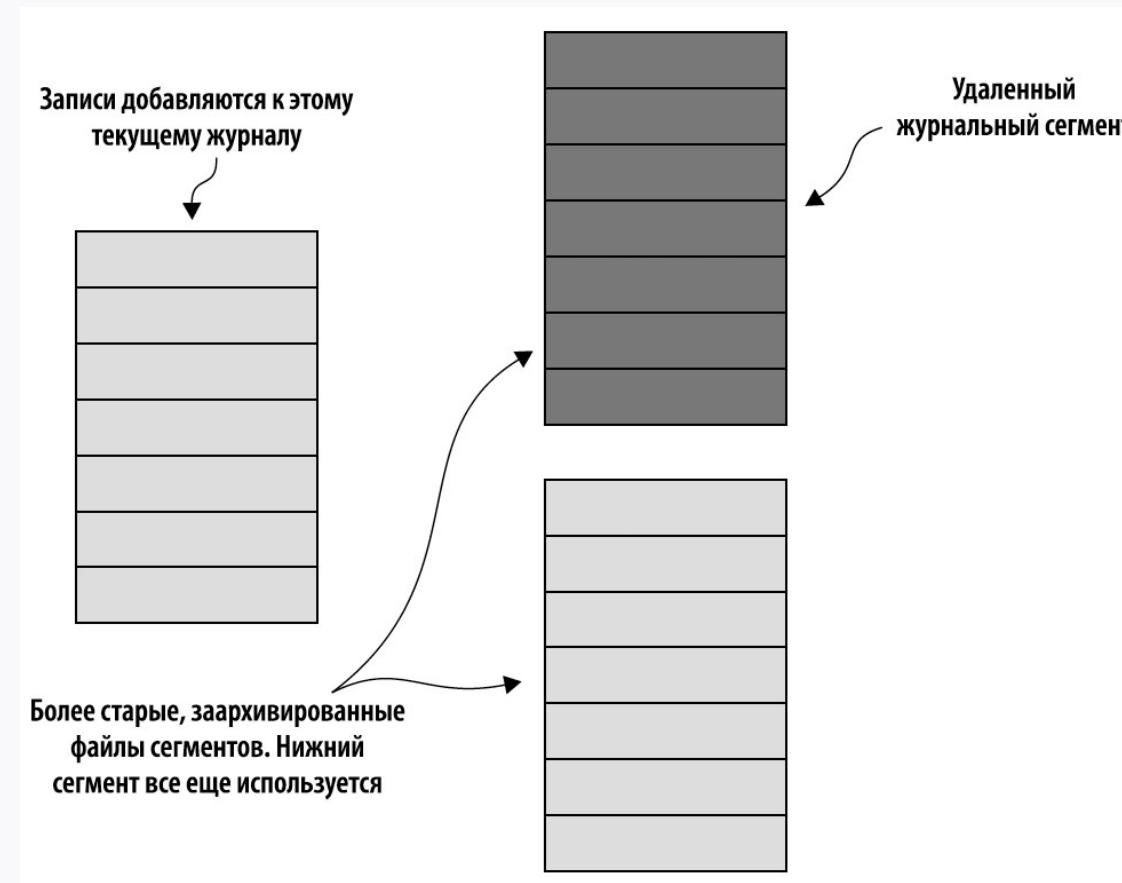
/logs/topicB_1

/logs/topicB_2

Удаление журналов

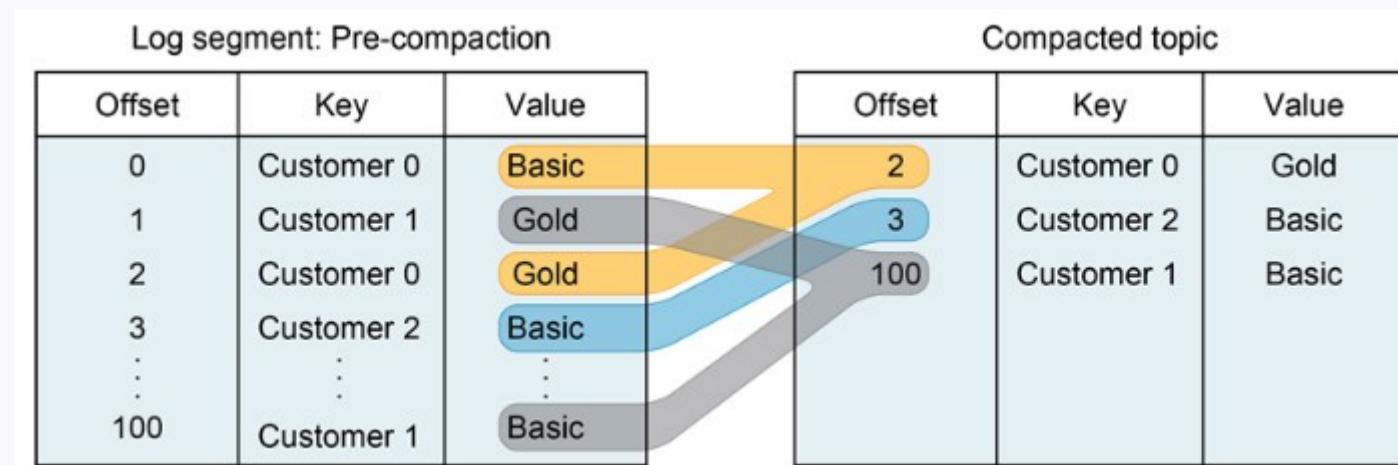
Двухэтапная стратегия удаления:

- архивация (roll) журналов в сегменты (`timestamp + log.roll.ms`)
- удаление старых сегментов

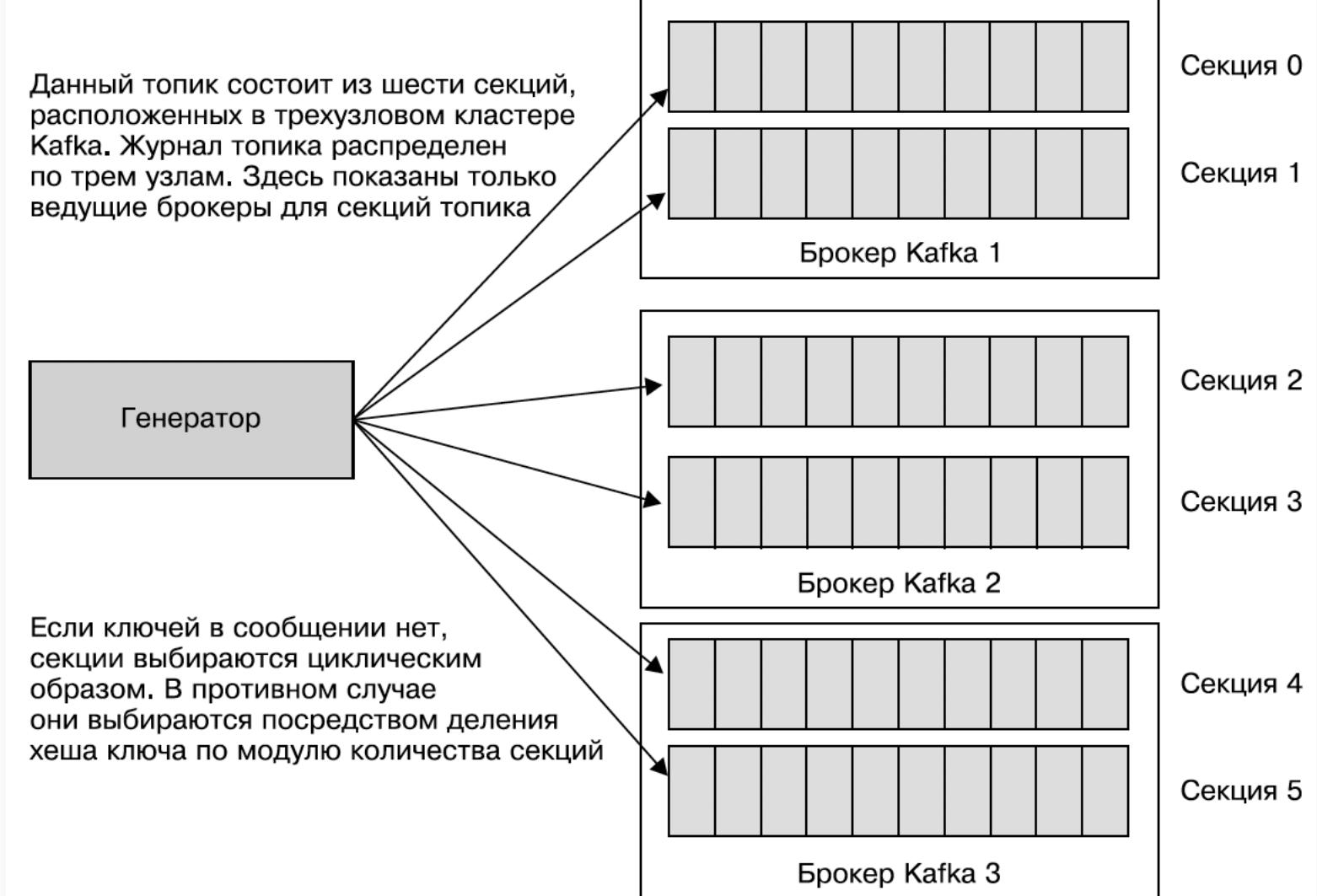


Сжатие журналов

- Kafka поддерживает удаление старых записей (сжатие) за счёт стратегии хранения:
 - *delete* (удалять) – удаление событий, чей возраст превышает срок хранения
 - *compact* (сжать) – сохранение только последних значений для каждого ключа
- Сжатие возможно только для тем с непустыми ключами

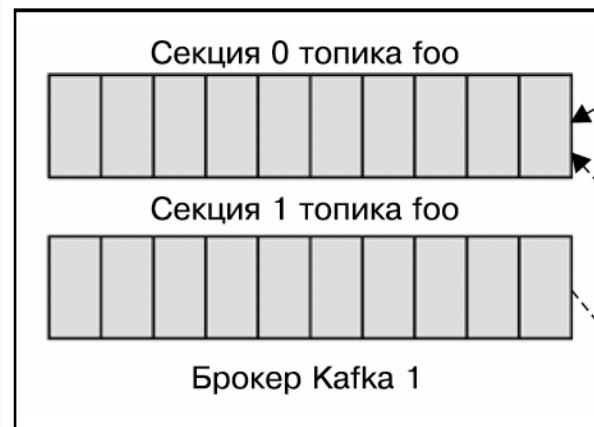


Распределённый журнал

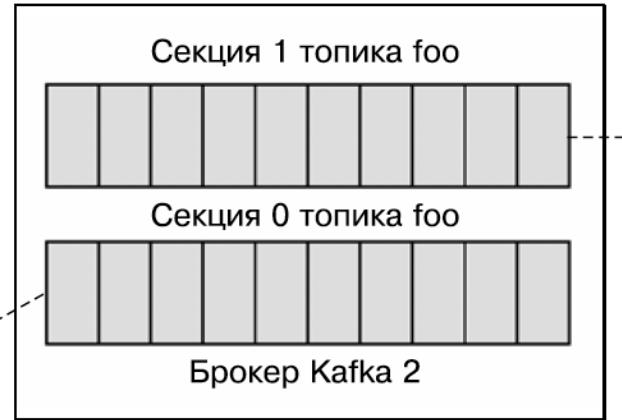


Репликация

Топик foo состоит из двух секций, его уровень репликации — 3. Штриховые линии между секциями указывают на ведущие брокеры данных секций. Генераторы записывают данные на ведущие брокеры, а ведомые брокеры читают данные с них



Брокер 1 — ведущий для секции 0 и ведомый для секции 1 на брокере 3



Брокер 2 — ведомый как для секции 0 на брокере 1, так и для секции 1 на брокере 3

Брокер 3 — ведомый для секции 0 на брокере 1 и ведущий для секции 1



Обязанности контроллера

Топик foo состоит из двух секций, его уровень репликации — 3. Изначально у него следующие ведущие и ведомые брокеры:

Брокер 1 — ведущий для секции 0

и ведомый для секции 1

Брокер 2 — ведомый для секции 0 и секции 1

Брокер 3 — ведомый для секции 0

и ведущий для секции 1

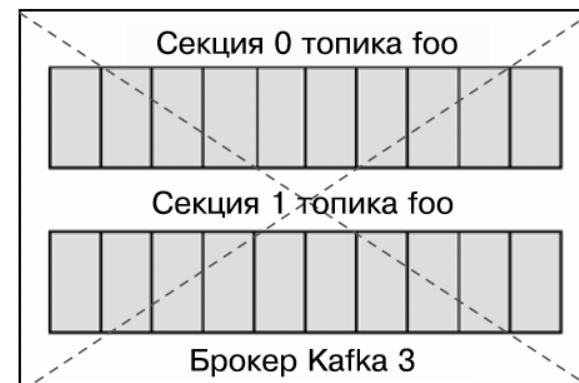
Брокер 3 перестал реагировать на контрольные сигналы ZooKeeper



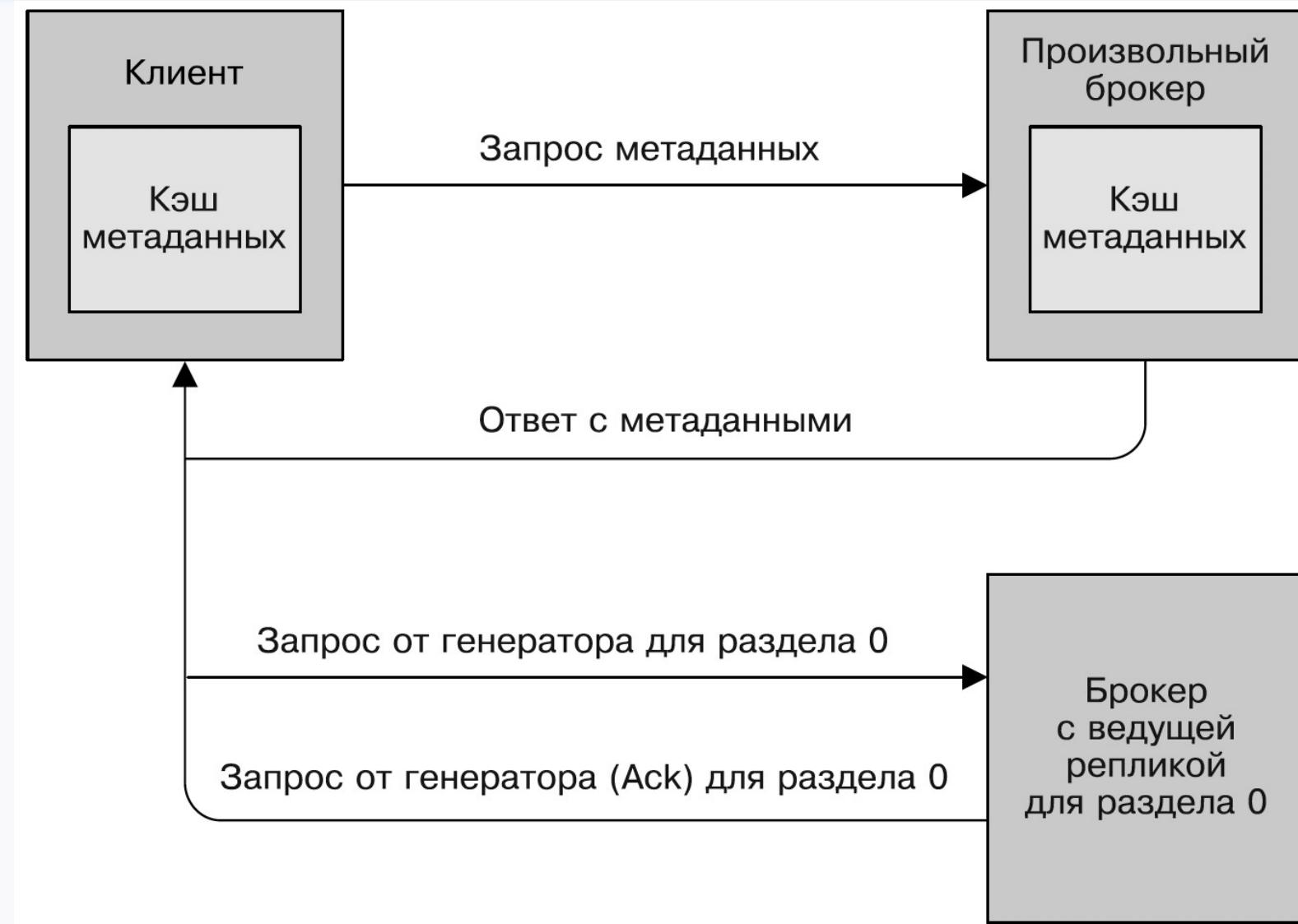
Шаг 1: будучи ведущим, брокер 1 обнаружил сбой брокера 3



Шаг 2: контроллер делает ведущим для секции 1 брокер 2 вместо брокера 3. Все записи секции 1 теперь будут попадать на брокер 2, а брокер 1 будет потреблять сообщения для секции 1 с брокера 2



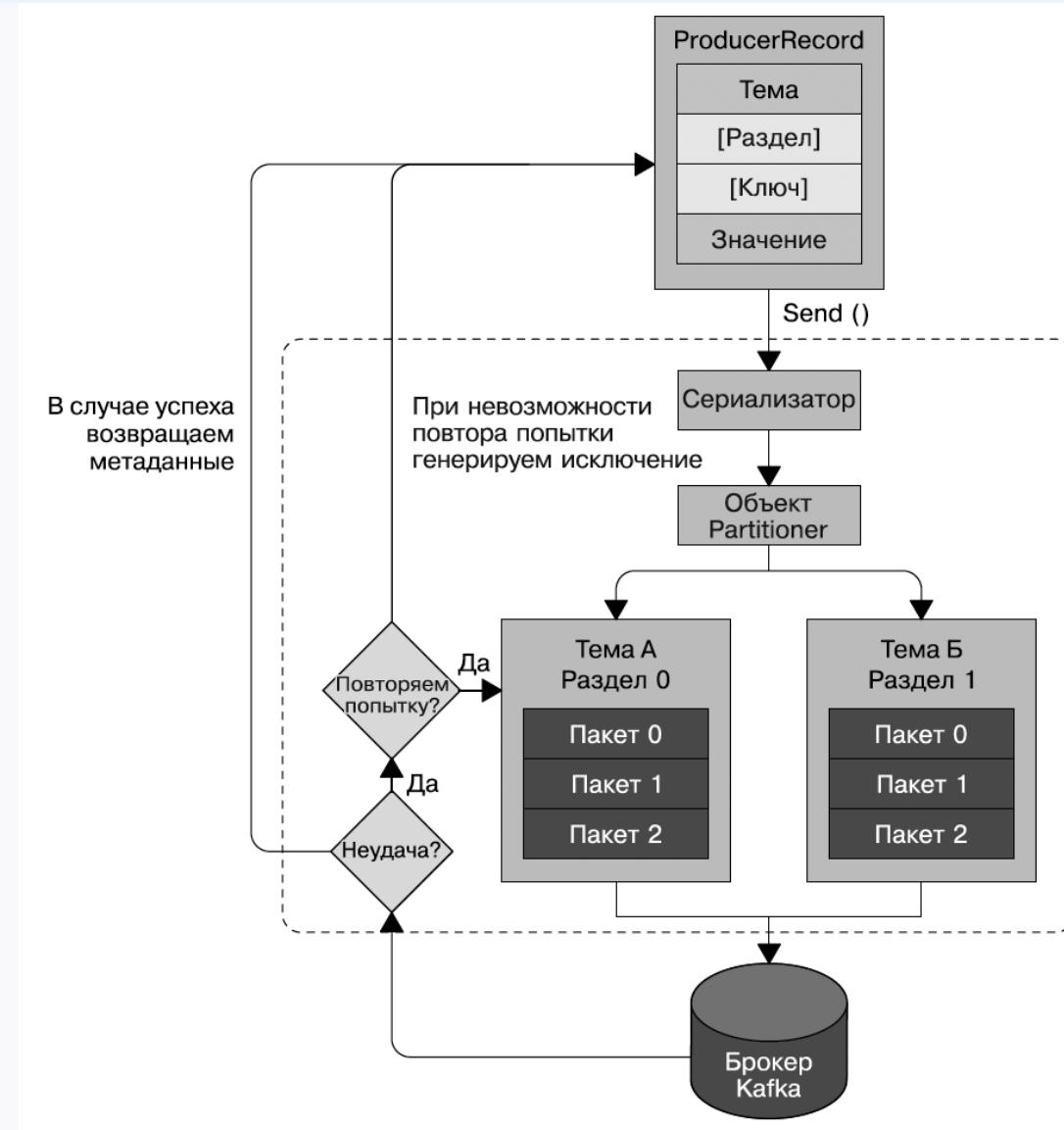
Маршрутизация запросов



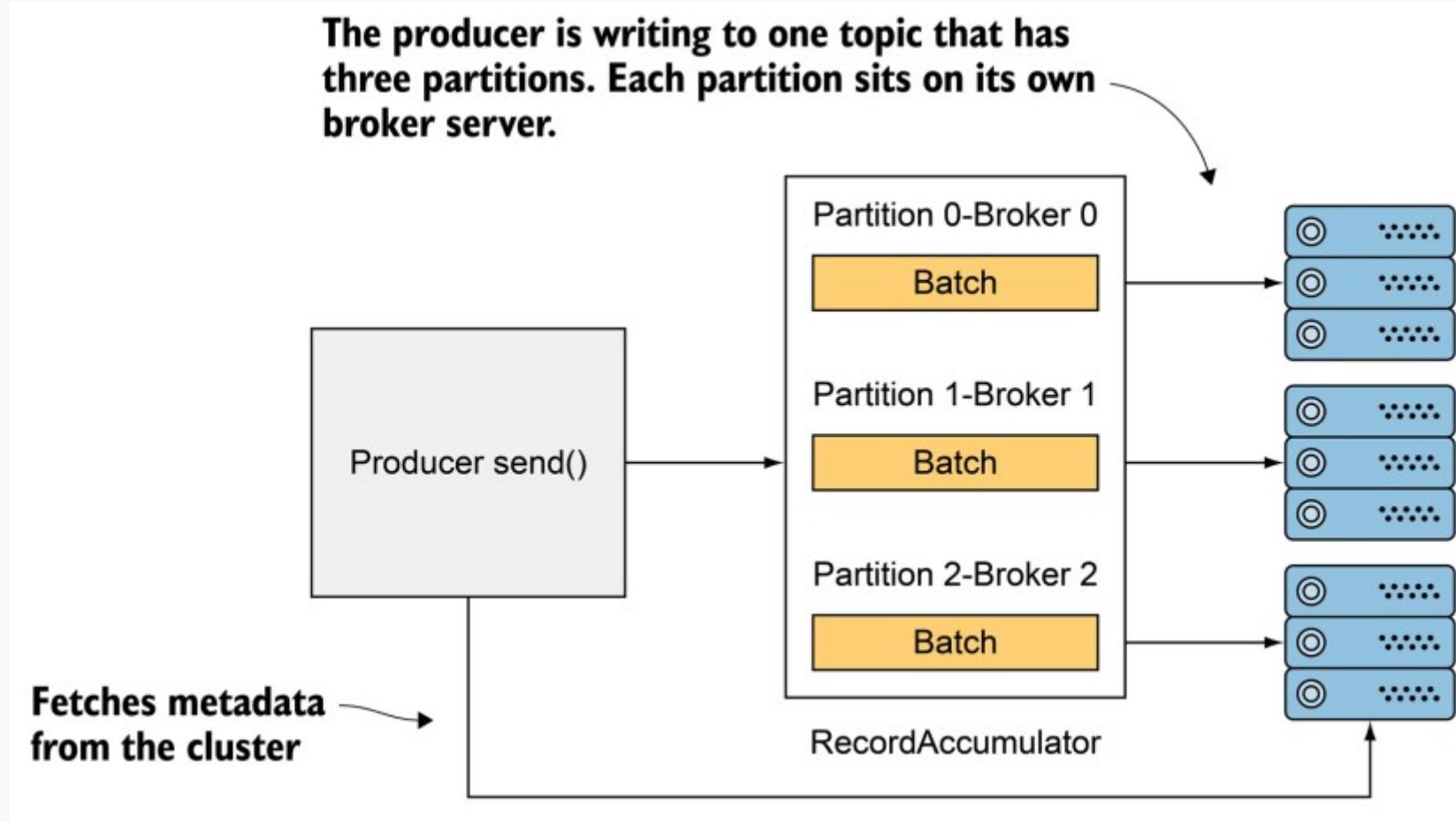
The background features a high-angle aerial photograph of a dense urban area with numerous skyscrapers. Overlaid on this image is a network visualization consisting of a central cluster of light blue dots connected by thin lines, which then radiate outwards across the entire frame.

Producer

Producer – запись сообщений в Kafka



Producer – запись сообщений в Kafka



Свойства Producer

Обязательные свойства:

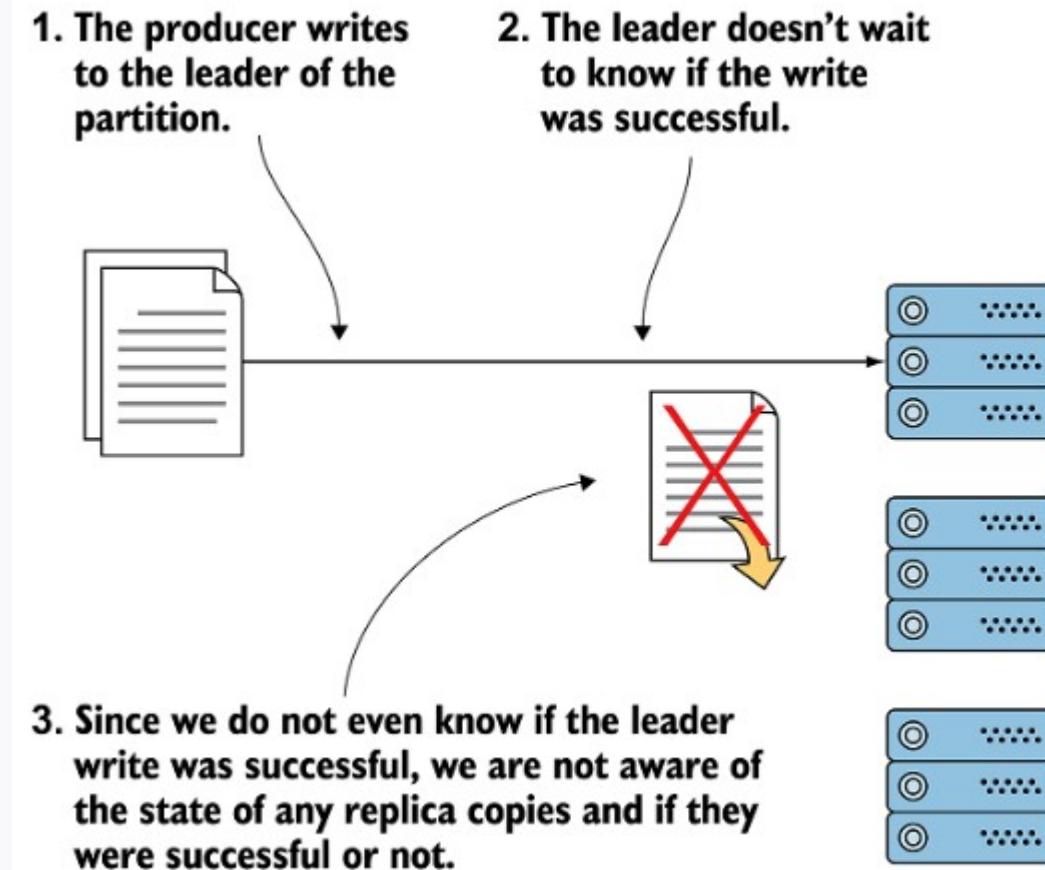
- `bootstrap.servers` – список пар `host:port` брокеров
- `key.serializer` – имя класса, используемого для сериализации ключей записей
- `value.serializer` – имя класса, используемого для сериализации значений записей

Необязательные свойства:

- `acks` – число получивших сообщение реплик секций, требующееся для признания производителем операции записи успешной.
 - `acks = 0` – Производитель не ждёт ответа от брокера, чтобы считать отправку успешной
 - `acks = 1` – Производитель получает ответ от брокера об успешном получении, как только ведущая реплика получит сообщение
 - `acks = all` – Производитель получает ответ от брокера об успешном получении, когда оно дойдёт до всех синхронизируемых реплик
- `retries` – число попыток отправить сообщение
- `partitioner.class` – класс для секционирования

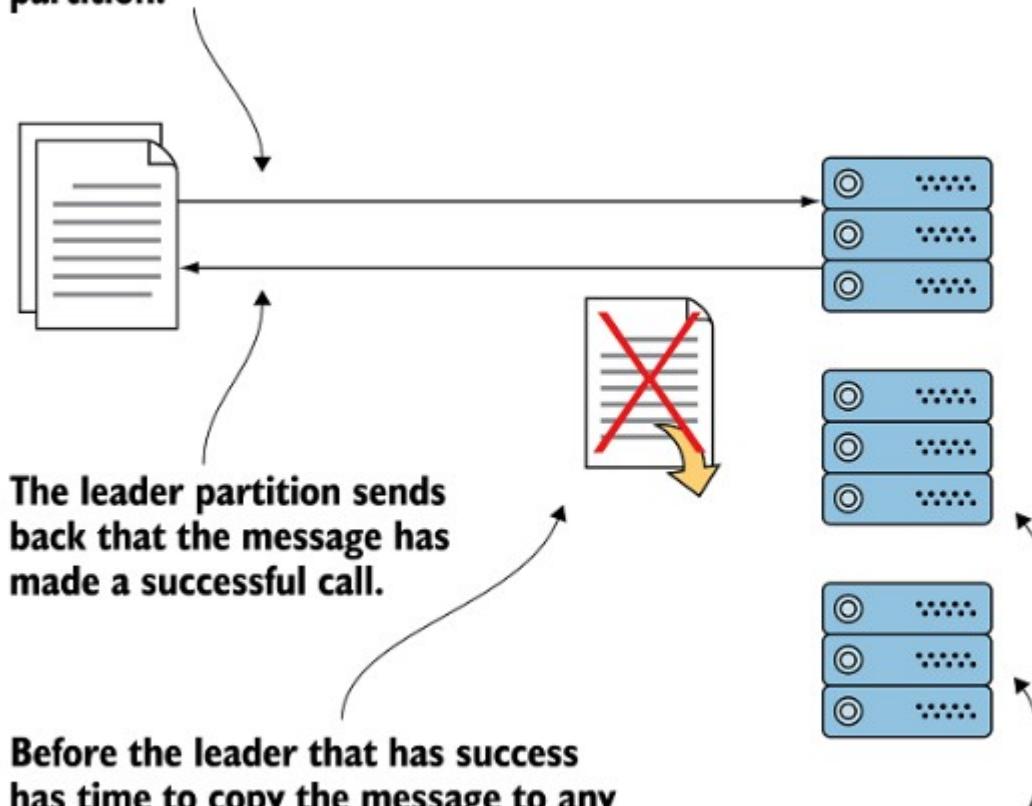
<http://kafka.apache.org/27/javadoc/org/apache/kafka/clients/producer/ProducerConfig.html>

Property acks=0



Property acks=1

1. The producer writes to the leader of the partition.

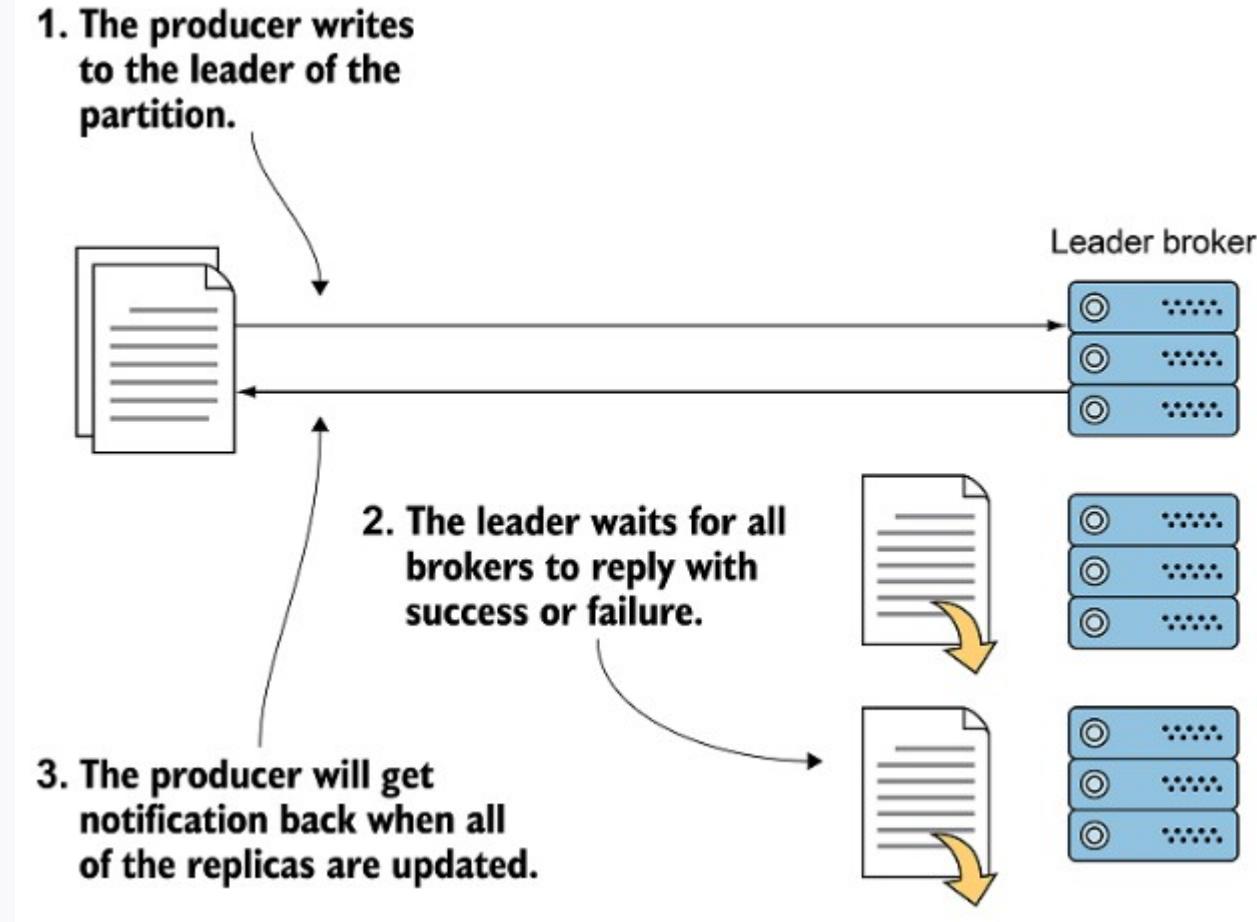


2. The leader partition sends back that the message has made a successful call.

3. Before the leader that has success has time to copy the message to any follower replicas, the leader fails. This means that the message could appear lost to the remaining brokers.

4. These brokers never see the message if though it was seen by the leader.

Property acks=all



Создание Producer

```
private Properties kafkaProps = new Properties(); ①  
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");  
  
kafkaProps.put("key.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer"); ②  
kafkaProps.put("value.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");  
  
producer = new KafkaProducer<String, String>(kafkaProps); ③
```

Оправка сообщений

Основные методы отправки сообщений:

- *Отправить и забыть*: отправляем и не проверяем статус доставки
- *Синхронная отправка*: после отправки получаем Future
- *Асинхронная отправка*: передаём функцию обратного вызова

Реализация пользовательской стратегии секционирования

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

    public void configure(Map<String, ?> configs) {} ❶

    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes, Cluster cluster)
    {
        List<PartitionInfo> partitions =
            cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceof String))) ❷
            throw new InvalidRecordException("We expect all messages
                to have customer name as key")

        if (((String) key).equals("Banana"))
            return numPartitions-1;
        // Banana всегда попадает в последний раздел
        // Другие записи распределяются по разделам путем хеширования
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1))
    }

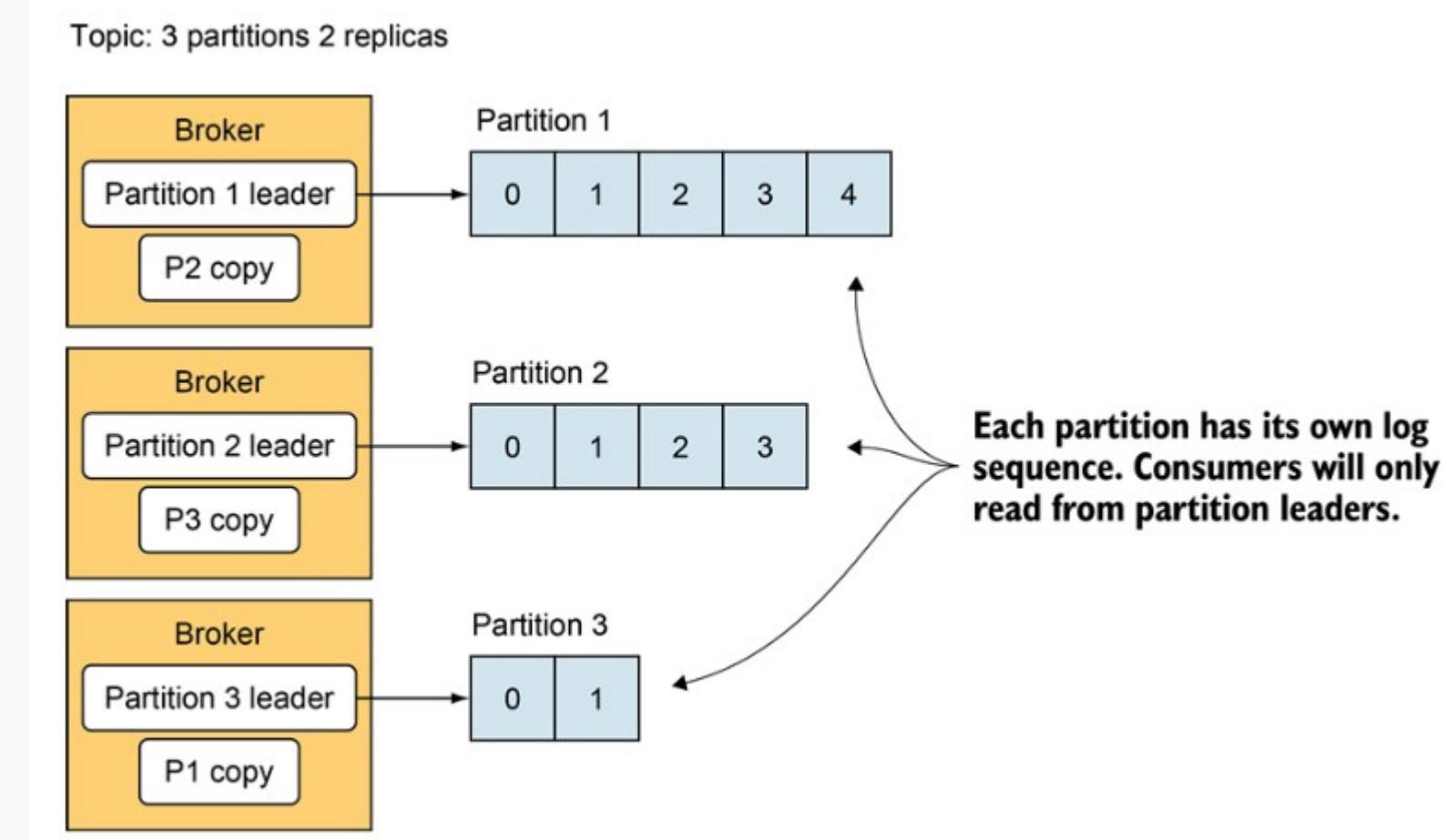
    public void close() {}

}
```

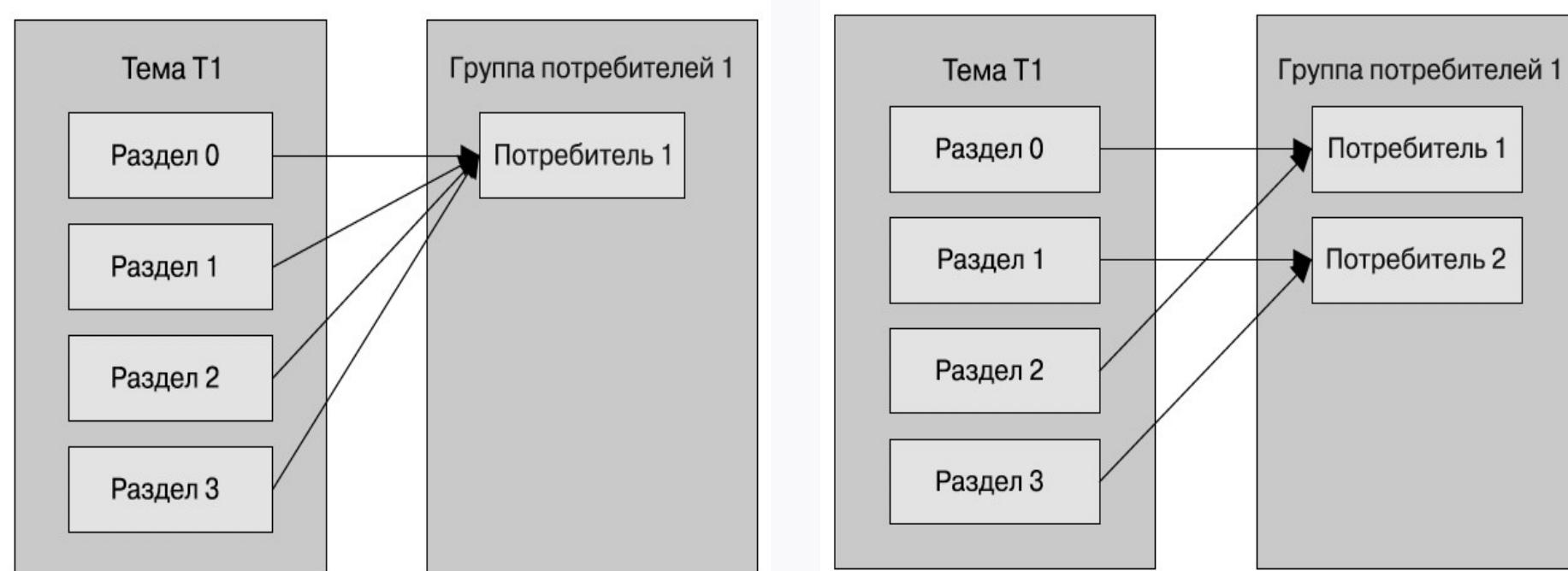


Consumer

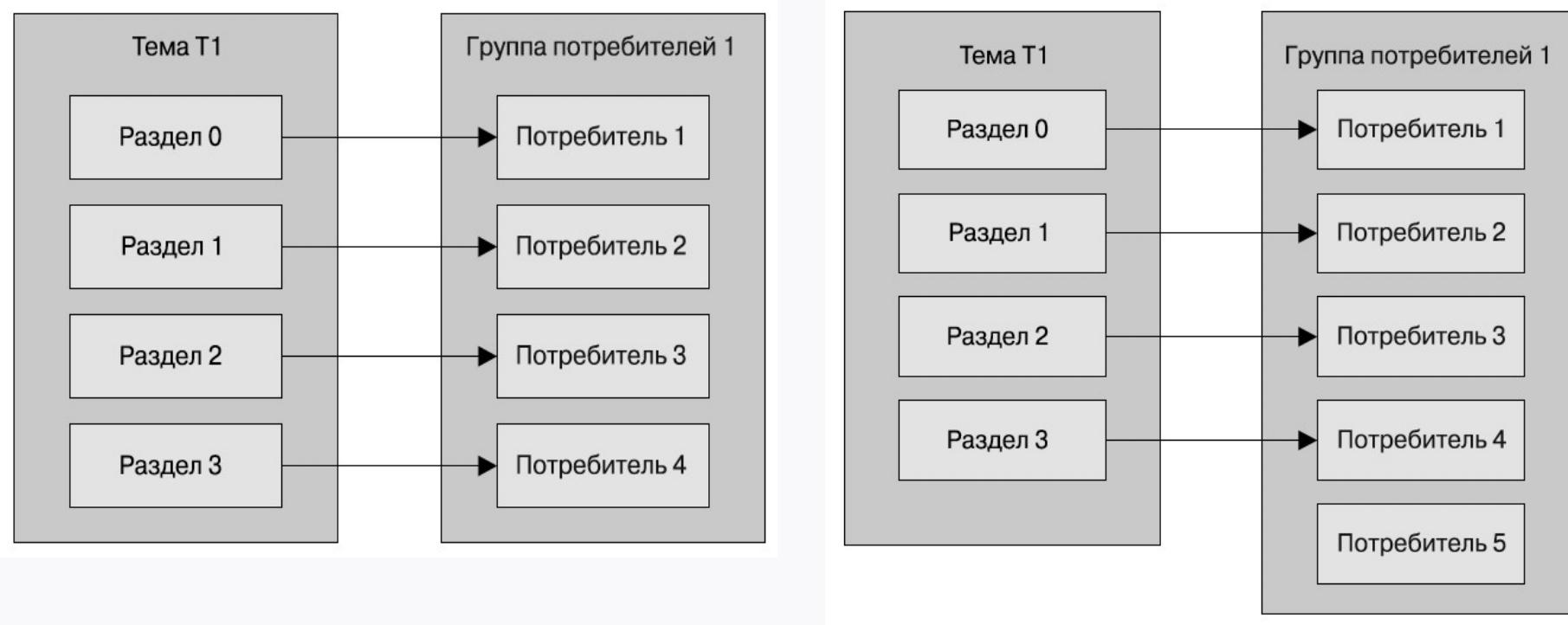
Consumer – чтение сообщений из Kafka



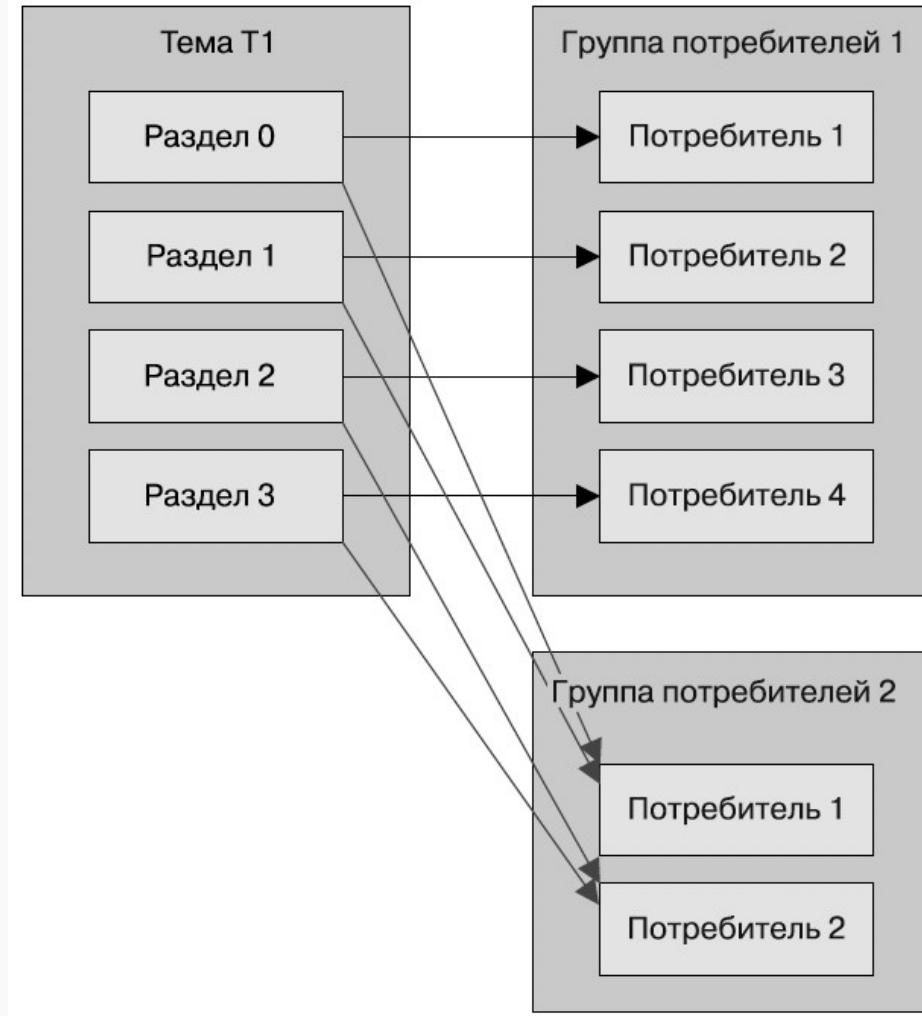
Consumer – чтение сообщений из Kafka



Consumer – чтение сообщений из Kafka



Consumer – чтение сообщений из Kafka



Свойства Consumer

Обязательные свойства:

- `bootstrap.servers` – список пар `host:port` брокеров
- `key.deserializer` – имя класса, используемого для десериализации ключей записей
- `value.deserializer` – имя класса, используемого для десериализации значений
- `group.id` – группа потребителей

Необязательные свойства:

- `auto.offset.reset` – поведение при начале чтения секции, если нет зафиксированного смещения:
 - `latest` – «самое поздние» (по умолчанию)
 - `earliest` – «самое раннее»
- `enable.auto.commit` – фиксировать ли смещение автоматически (`true` по умолчанию)
- `auto.commit.interval.ms` – интервал в мс для автоматической отправки смещений

<http://kafka.apache.org/27/javadoc/org/apache/kafka/clients/consumer/ConsumerConfig.html>

Создание Consumer

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(props);
```

Подписка на темы

Следующий шаг – подписка на одну или несколько тем:

```
consumer.subscribe(Collections.singletonList("customerCountries"));
```

Можно использовать регулярные выражения:

```
consumer.subscribe(Pattern.compile("test.*"));
```

Можно подписаться на конкретные темы и секции:

```
TopicPartition fooTopicPartition_0 = new TopicPartition("foo", 0);
TopicPartition barTopicPartition_0 = new TopicPartition("bar", 0);

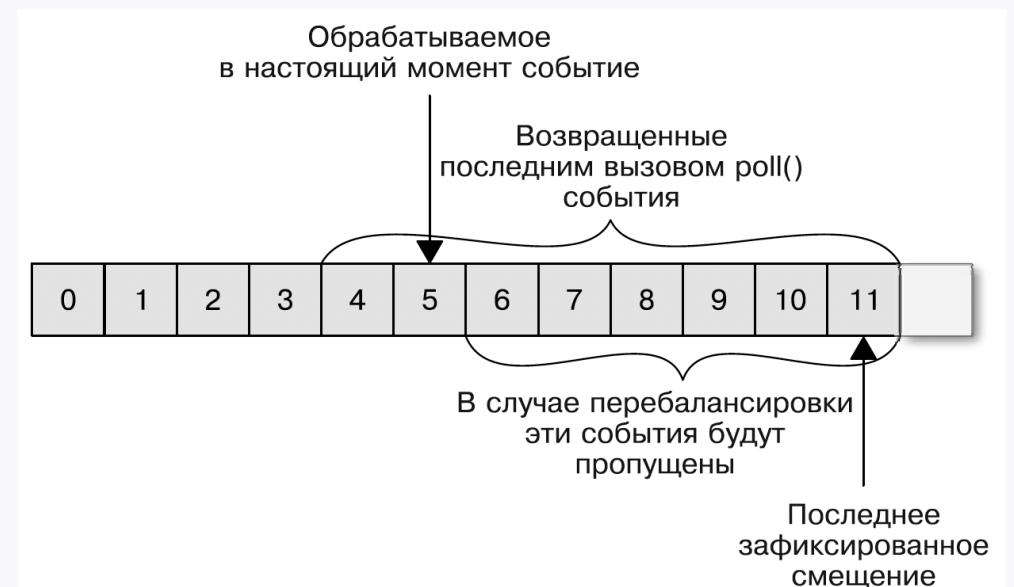
consumer.assign(Arrays.asList(fooTopicPartition_0, barTopicPartition_0));
```

Недостатки:

- Нет автоматического переназначения секций в случае сбоя
- Для фиксации придётся задать уникальный идентификатор группы

Фиксация и смещение

- Брокер Kafka не отслеживает подтверждения от потребителей
- Потребители могут использовать Kafka для отслеживания позиции (*смещение*) в секциях
- *Фиксация (commit)* – действие по обновлению текущей позиции потребителя в секции
- Потребители отправляют в специальную тему `__consumer_offsets` сообщение, содержащее смещение для каждой секции
- Аварийный сбой потребителя или присоединение нового потребителя инициирует *перебалансировку*



Прослушивание на предмет перебалансировки

Работа во время смены (добавления, удаления) принадлежности секций потребителю.

Передать в `subscribe()` объект `ConsumerRebalanceListener`:

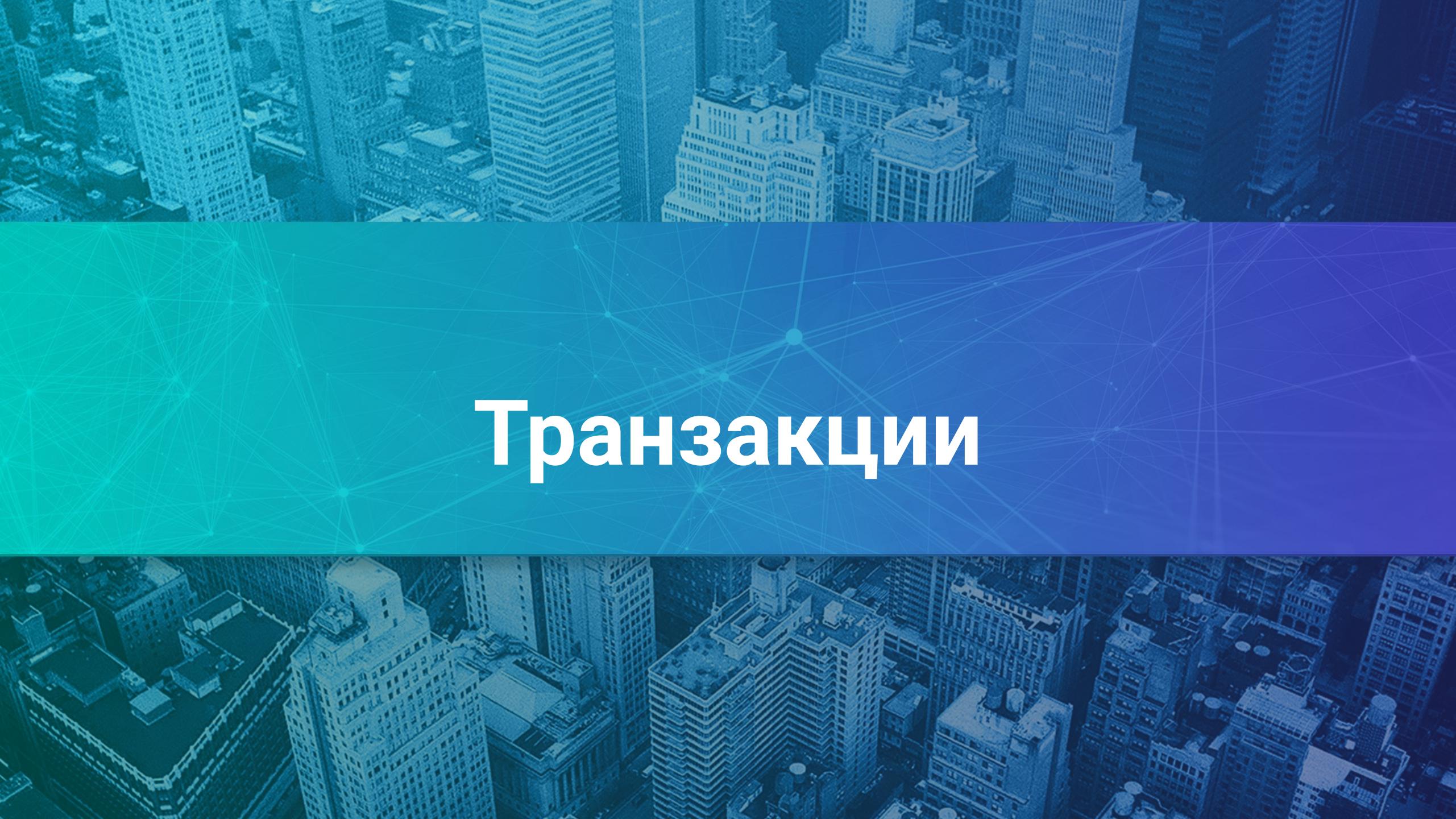
- `public void onPartitionsRevoked(Collection<TopicPartition> partitions)` – вызывается до начала перебалансировки и после завершения получения сообщения. Здесь необходимо фиксировать смещения для следующего потребителя.
- `public void onPartitionsAssigned(Collection<TopicPartition> partitions)` – вызывается после переназначения секций потребителю, но до того, как он начнёт получать сообщения.

Получение записей с заданным смещением

- `seekToBeginning` – чтение с начала
- `seekToEnd` – чтение с последнего
- `seek` – чтение с произвольного

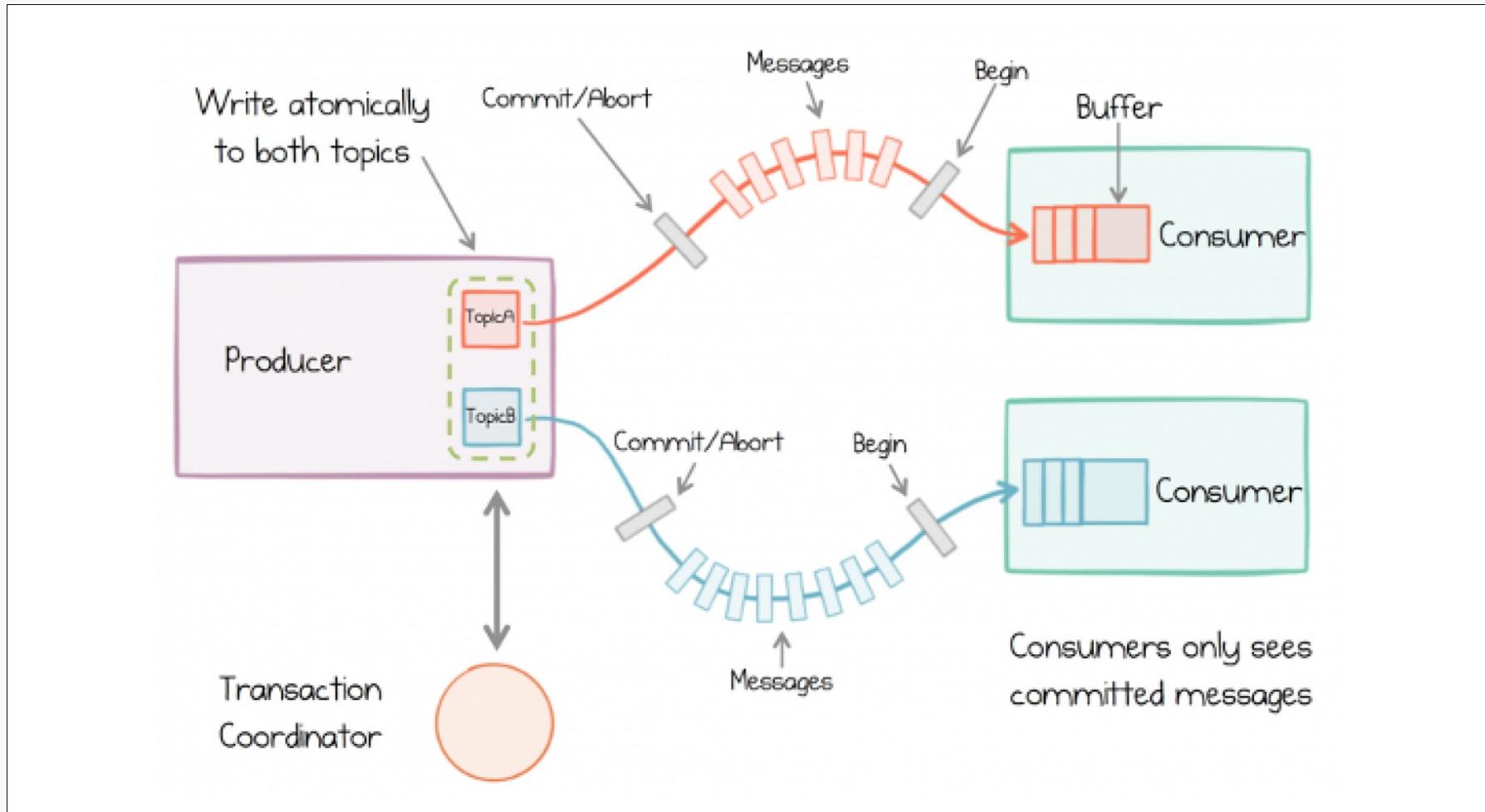
```
public class SaveOffsetsOnRebalance implements  
    ConsumerRebalanceListener {  
  
    public void onPartitionsRevoked(Collection<TopicPartition>  
        partitions) {  
        commitDBTransaction(); ①  
    }  
  
    public void onPartitionsAssigned(Collection<TopicPartition>  
        partitions) {  
        for(TopicPartition partition: partitions)  
            consumer.seek(partition, getOffsetFromDB(partition)); ②  
    }  
}  
  
consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));  
consumer.poll(0);  
  
for (TopicPartition partition: consumer.assignment())  
    consumer.seek(partition, getOffsetFromDB(partition)); ③
```

```
while (true) {  
    ConsumerRecords<String, String> records =  
        consumer.poll(100);  
    for (ConsumerRecord<String, String> record : records)  
    {  
        processRecord(record);  
        storeRecordInDB(record);  
        storeOffsetInDB(record.topic(), record.partition(),  
            record.offset()); ④  
    }  
    commitDBTransaction();  
}
```



Транзакции

Транзакции



Транзакции, Запись

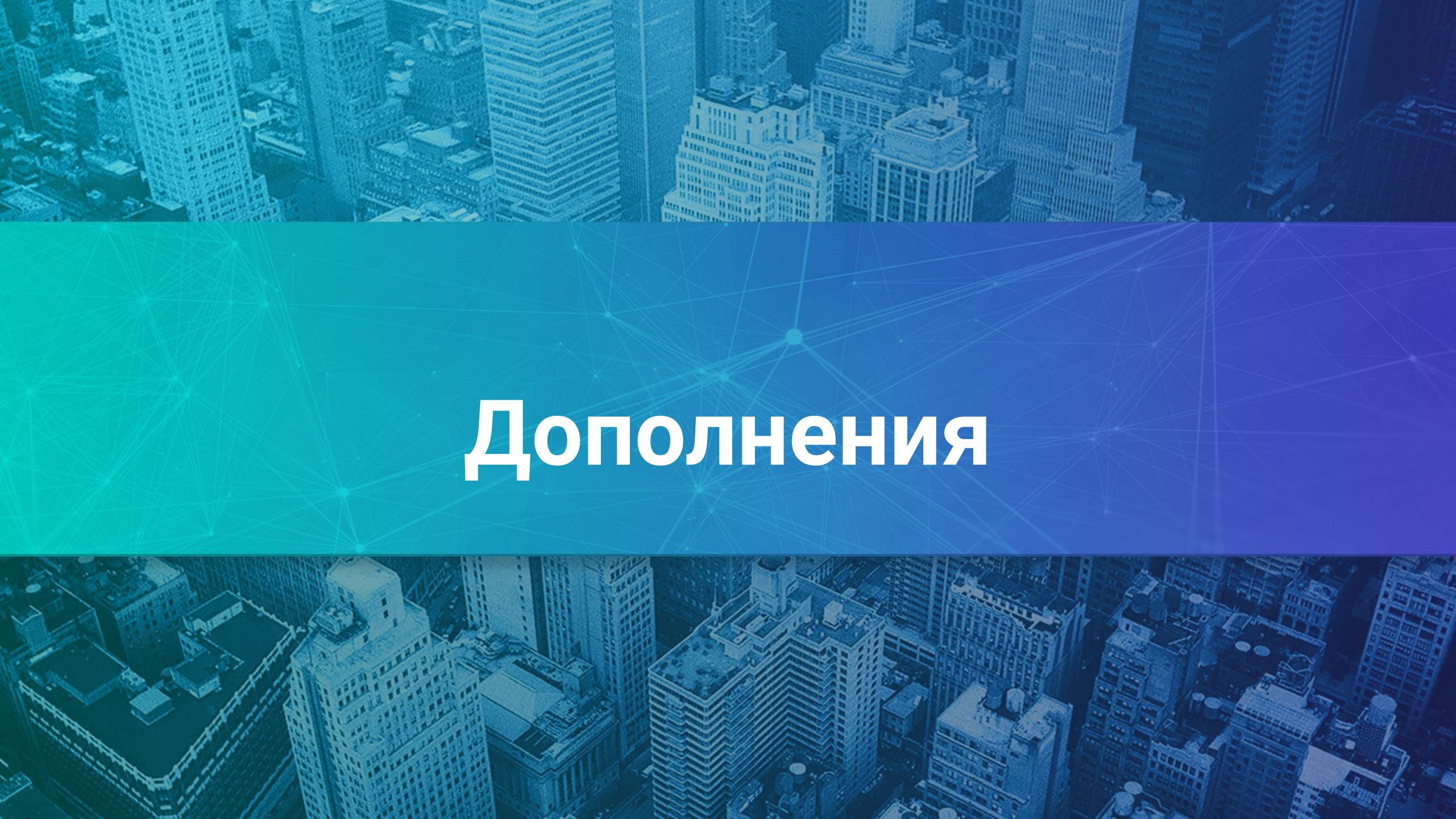
```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());

producer.initTransactions();

try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>"my-topic", Integer.toString(i), Integer.toString(i));
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // We can't recover from these exceptions, so our only option is to close the producer and exit.
    producer.close();
} catch (KafkaException e) {
    // For all other exceptions, just abort the transaction and try again.
    producer.abortTransaction();
}
producer.close();
```

Транзакции, Чтение

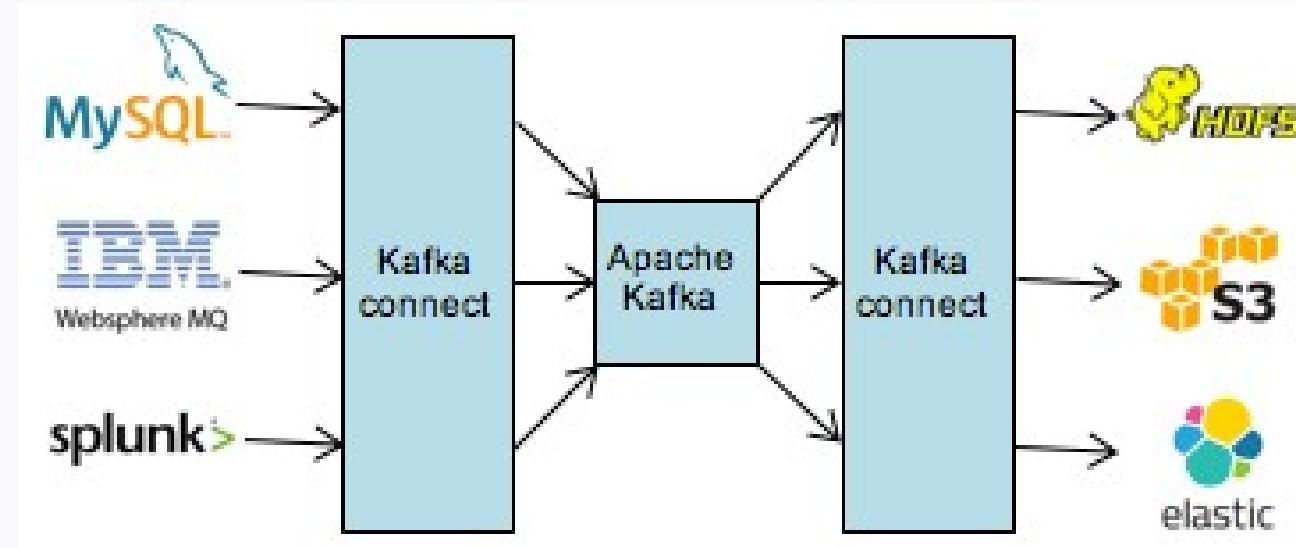
- Установить свойство:
 - `isolation.level = read_committed` – чтение только подтверждённых сообщений
- LSO (Last Stable Offset) – смещение первого сообщения открытой транзакции
- `read_committed` Consumer будет читать только до LSO и отфильтровывать сообщения из неподтверждённых транзакций
- `seekToEnd` и `endOffsets` возвращают LSO



Дополнения

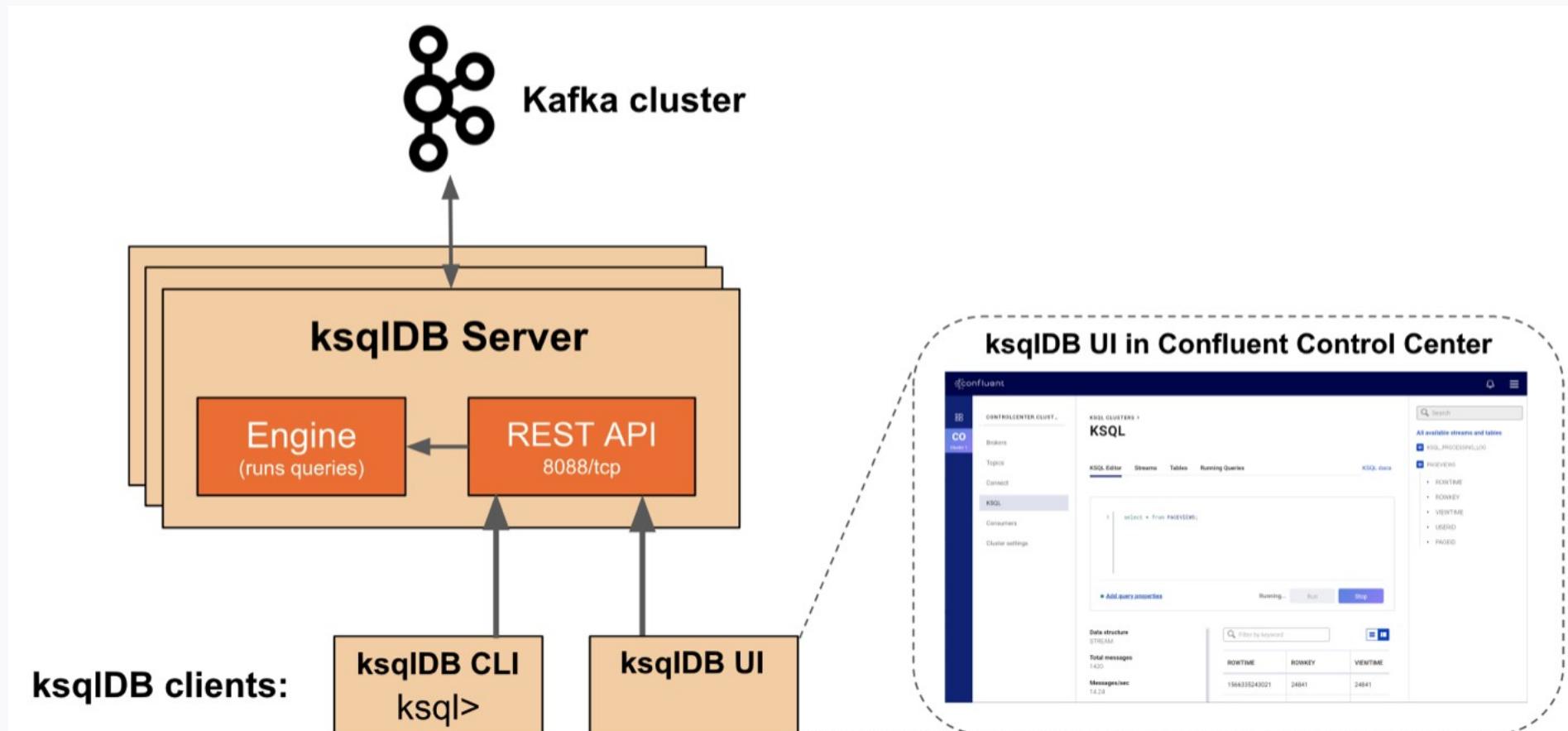
Kafka Connect

- Фреймворк и сервер приложений для интеграции Kafka с другими системами.
- Выделенный (standalone) и распределённые (distributed) режимы
- REST интерфейс
- Автоматическое управление смещениями
- Распределённый и масштабируемый по умолчанию
- Интеграция потоков (streaming) и пакетов (batch)



ksqldb

База данных потоковой обработки



ksqlDB, демо

<https://ksqldb.io/quickstart.html>

- CREATE STREAM riderLocations (profileId VARCHAR, latitude DOUBLE, longitude DOUBLE) WITH (kafka_topic='locations', value_format='json', partitions=1);
- SELECT * FROM riderLocations WHERE GEO_DISTANCE(latitude, longitude, 37.4133, -122.1162) <= 5 EMIT CHANGES;
- INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('c2309eec', 37.7877, -122.4205);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('18f4ea86', 37.3903, -122.0643);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('4ab5cbad', 37.3952, -122.0813);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('8b6eae59', 37.3944, -122.0813);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('4a7c7b41', 37.4049, -122.0822);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES ('4ddad000', 37.7857, -122.4011);

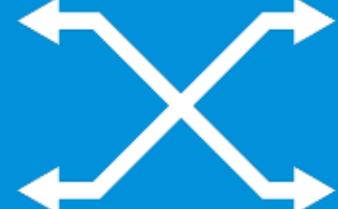
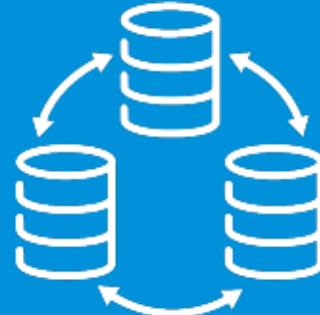


RabbitMQ Streams

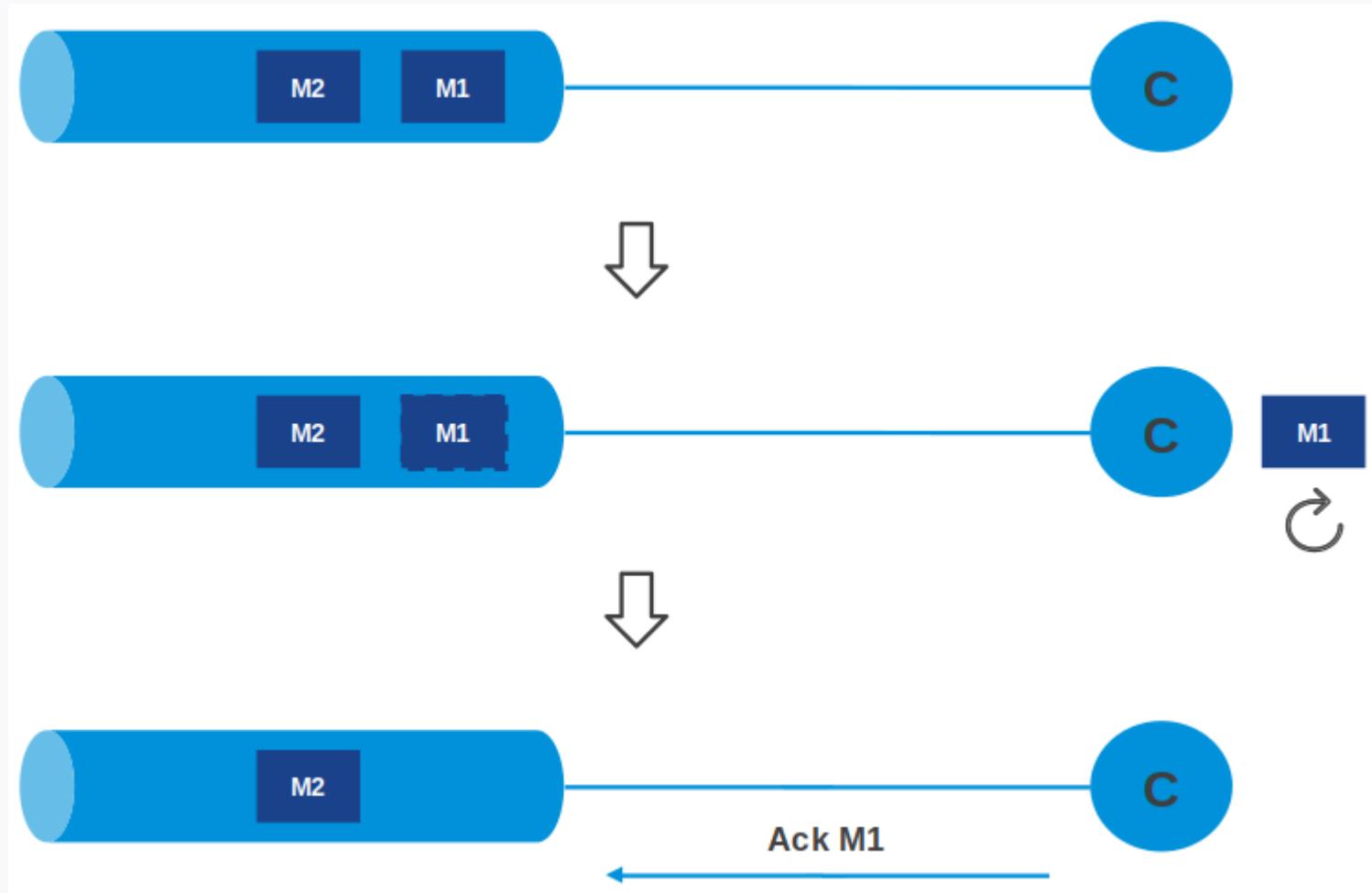
RabbitMQ Streams

Новый тип структуры данных в RabbitMQ

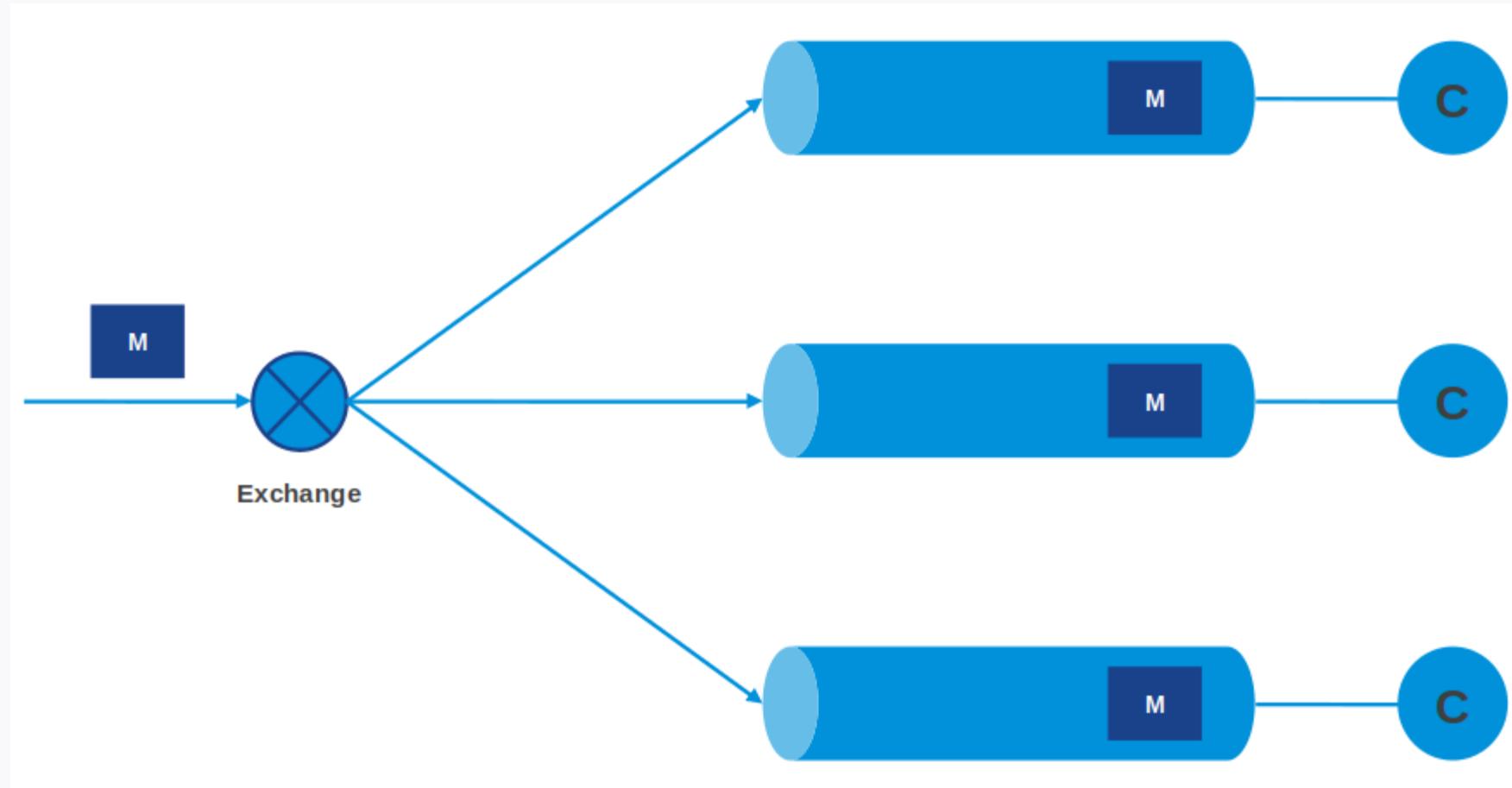
- Моделирует журнал только для добавления
- Постоянные и реплицируемые
- Неразрушающая семантика потребления
- AMQP 0.9.1 и новый протокол



Чтение в традиционных очередях деструктивно



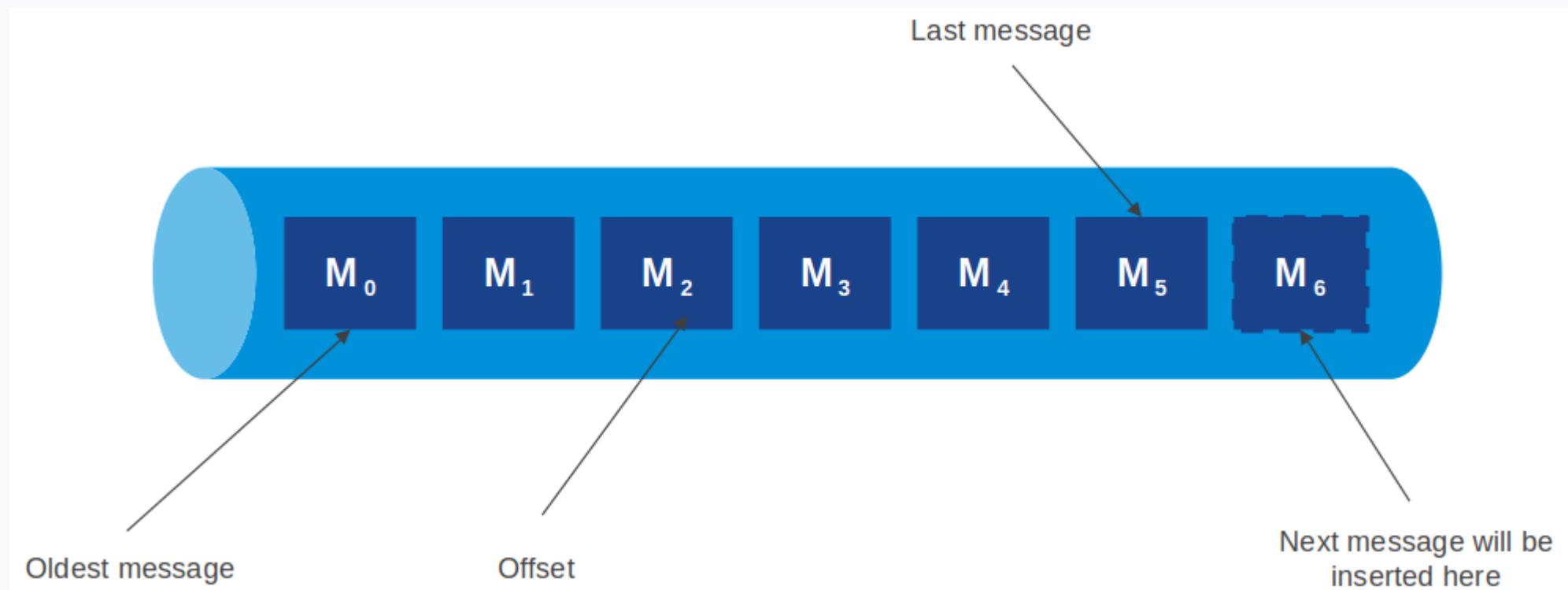
Fan-out в традиционных очередях требует много очередей



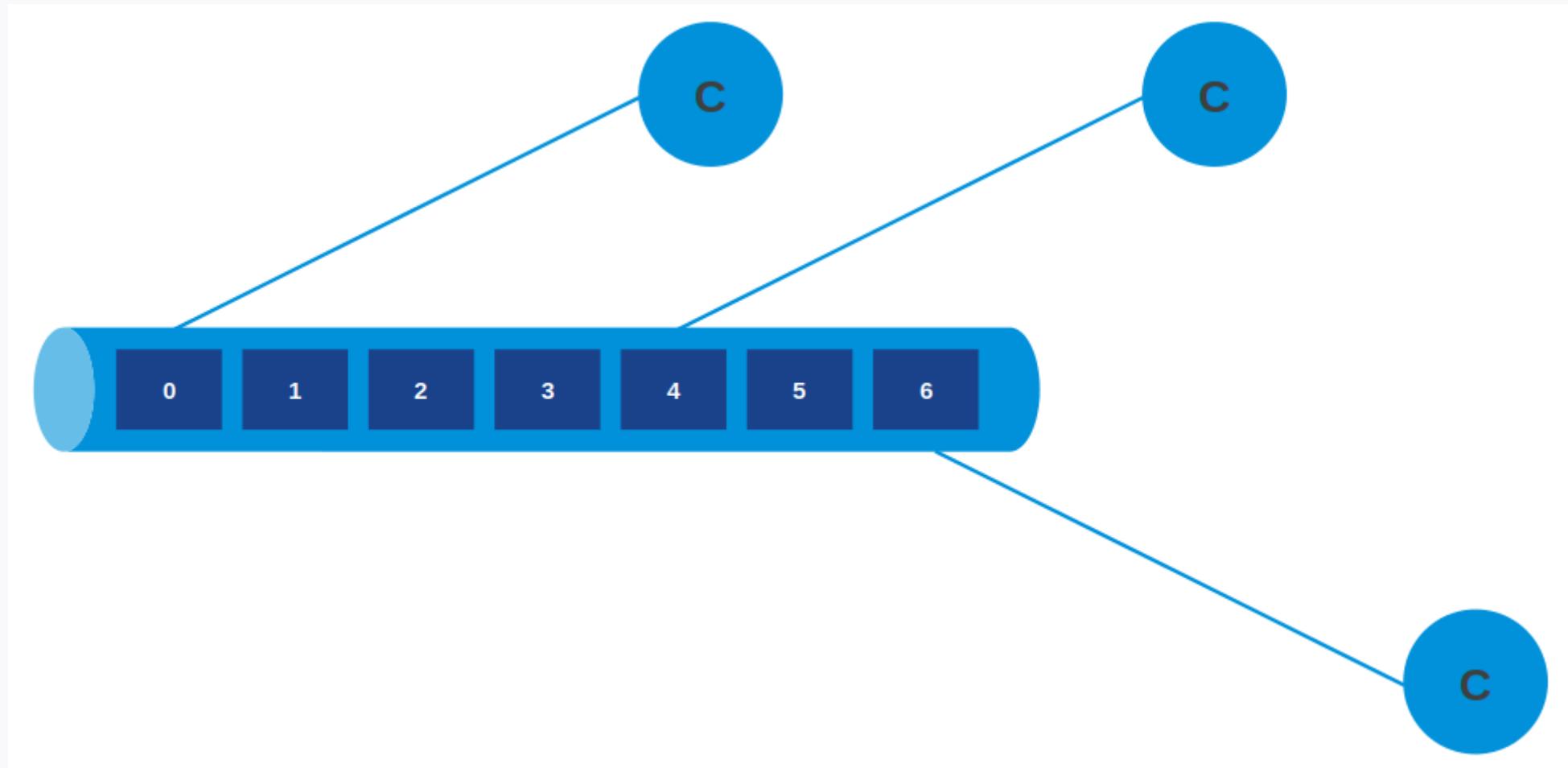
Абстракция журнала

Поток моделирует журнал только для добавления:

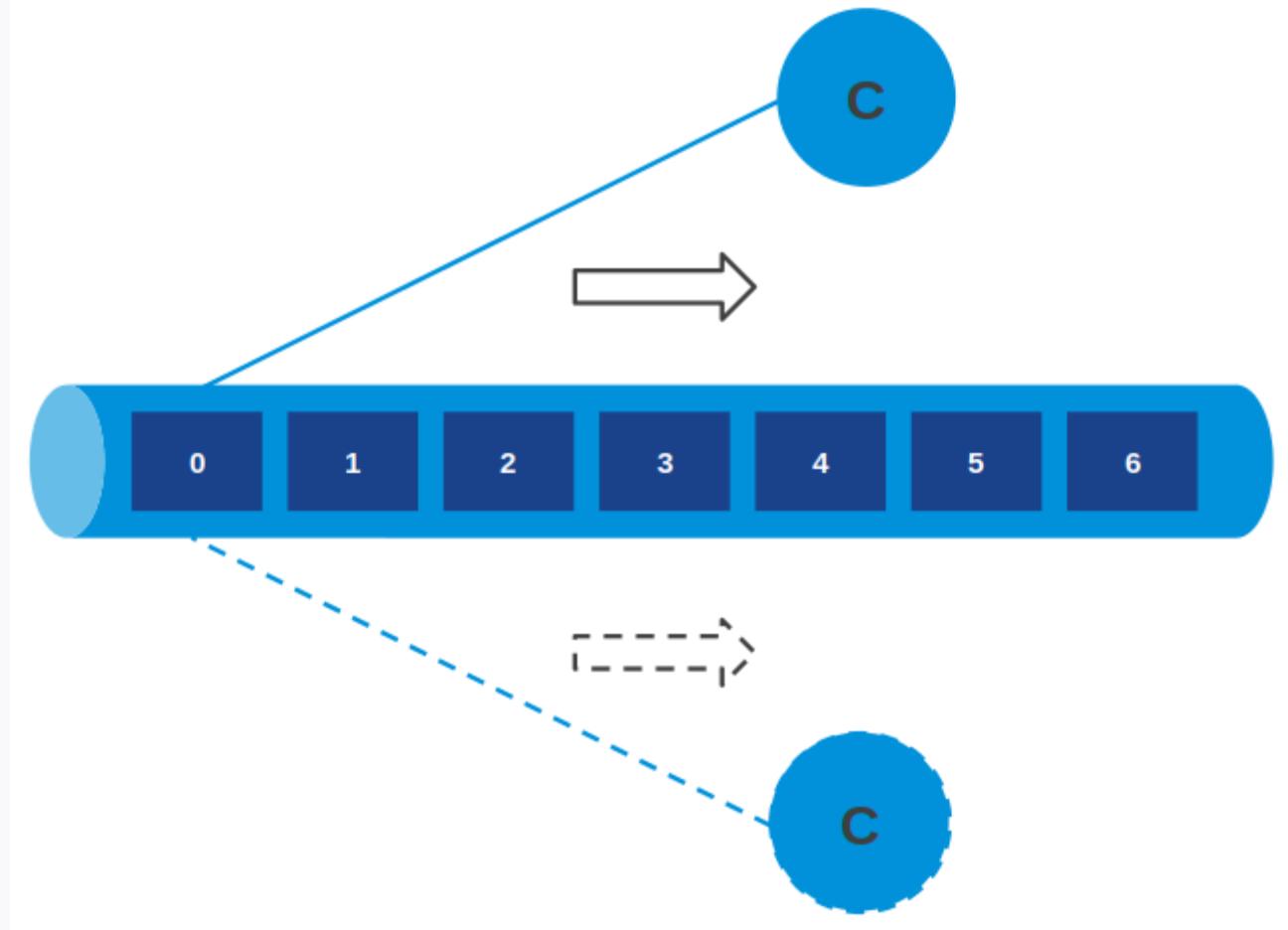
- Структура данных FIFO
- Никакого разрушительного чтения



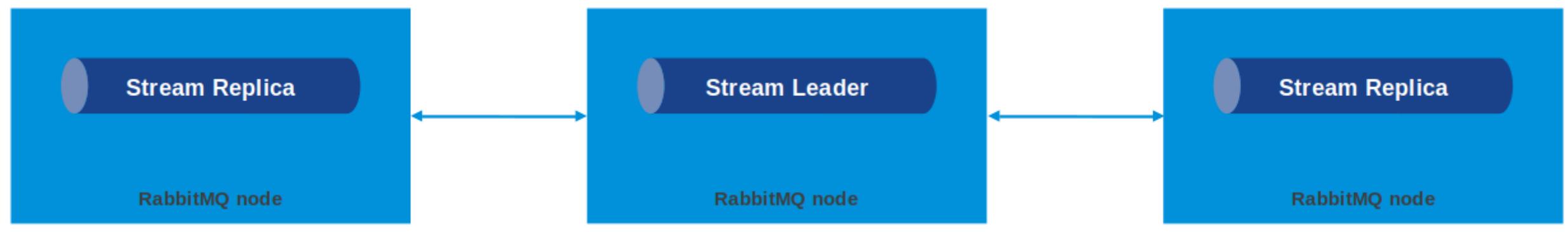
Любое количество потребителей может читать из потока



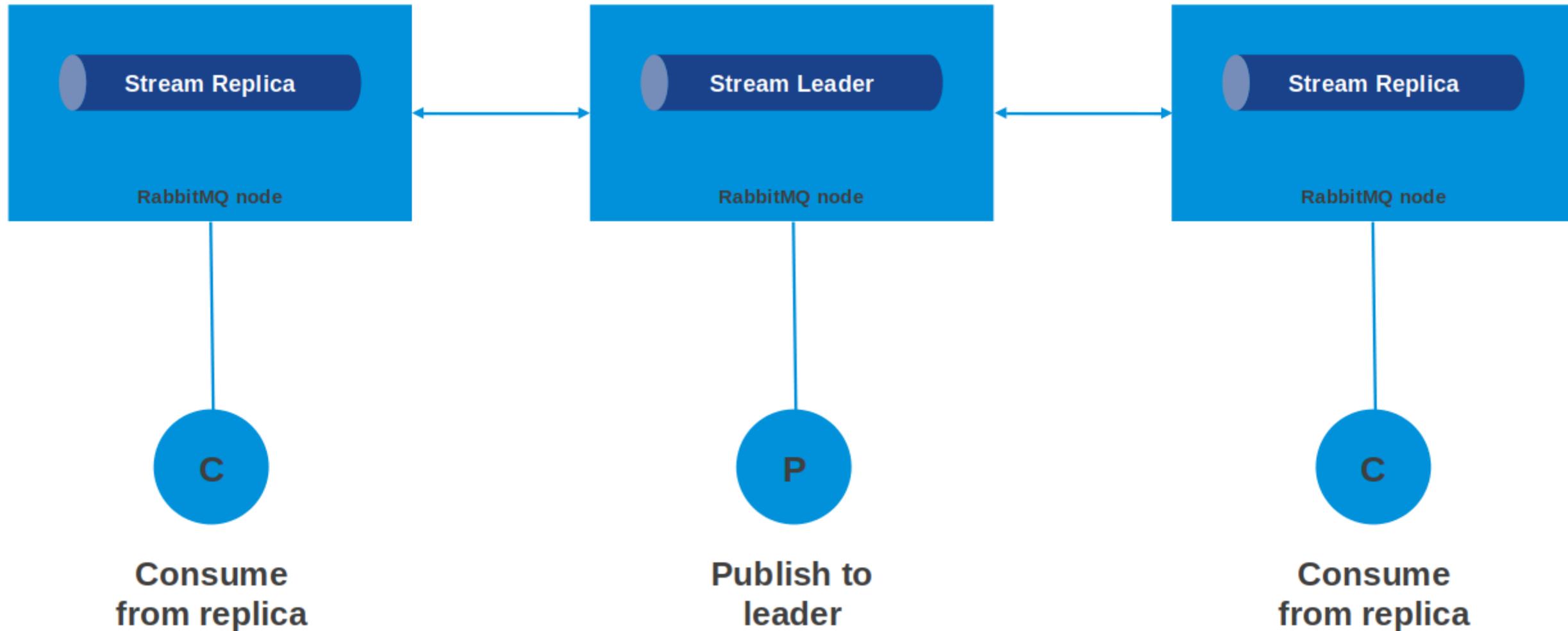
Потребитель может прочитать поток несколько раз



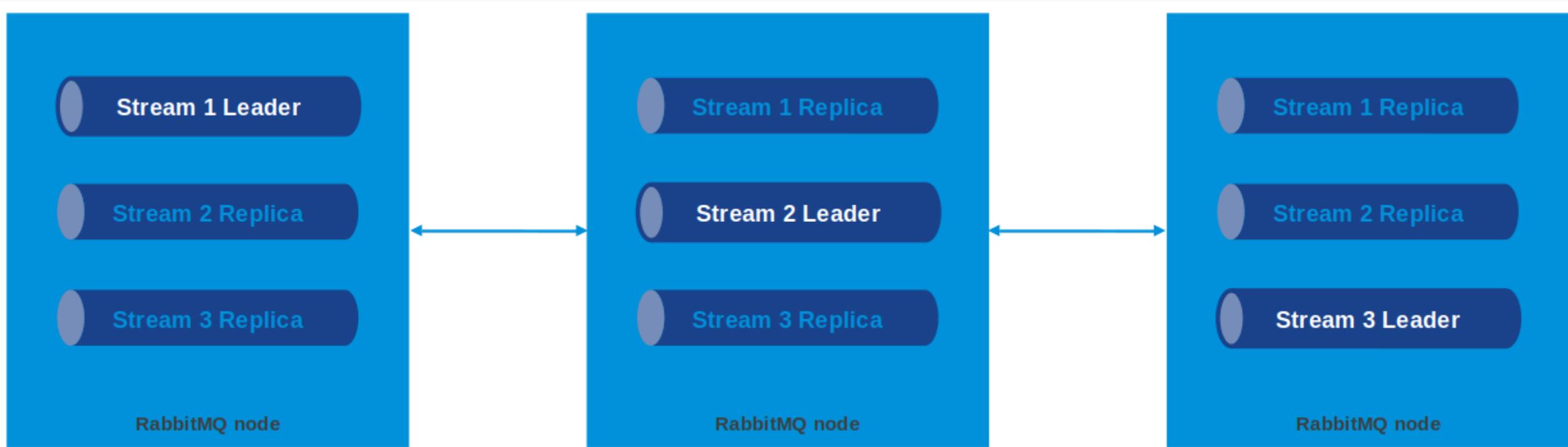
Поток является постоянным и реплицируемым



Издатели и потребители

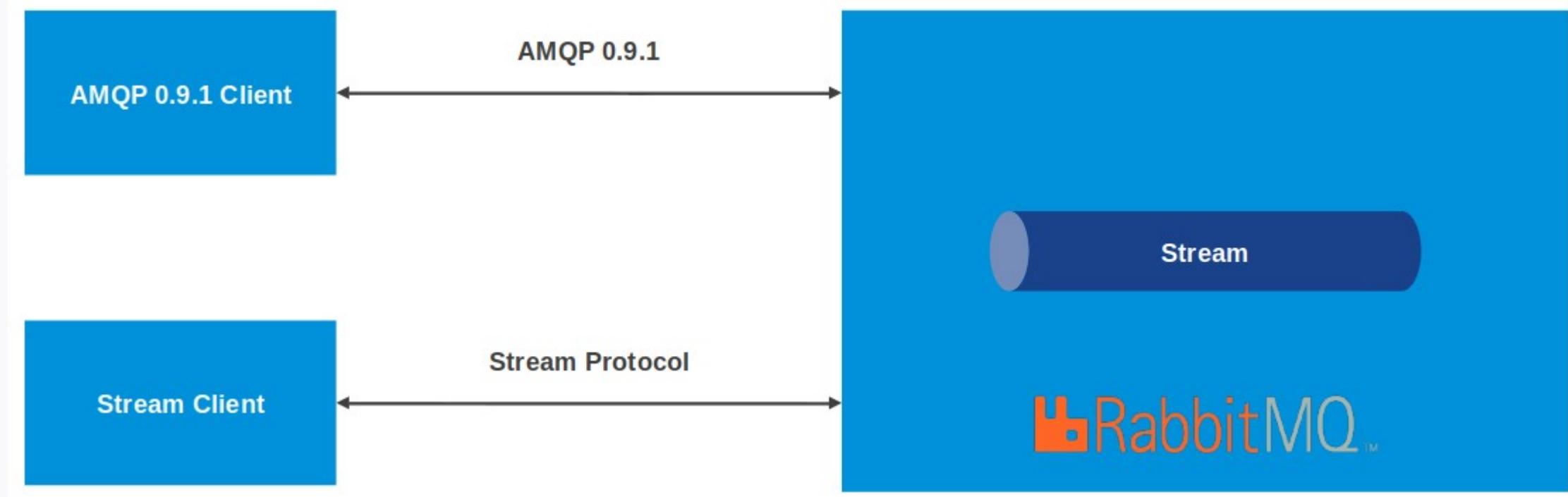


Потоки распространяются по всему кластеру

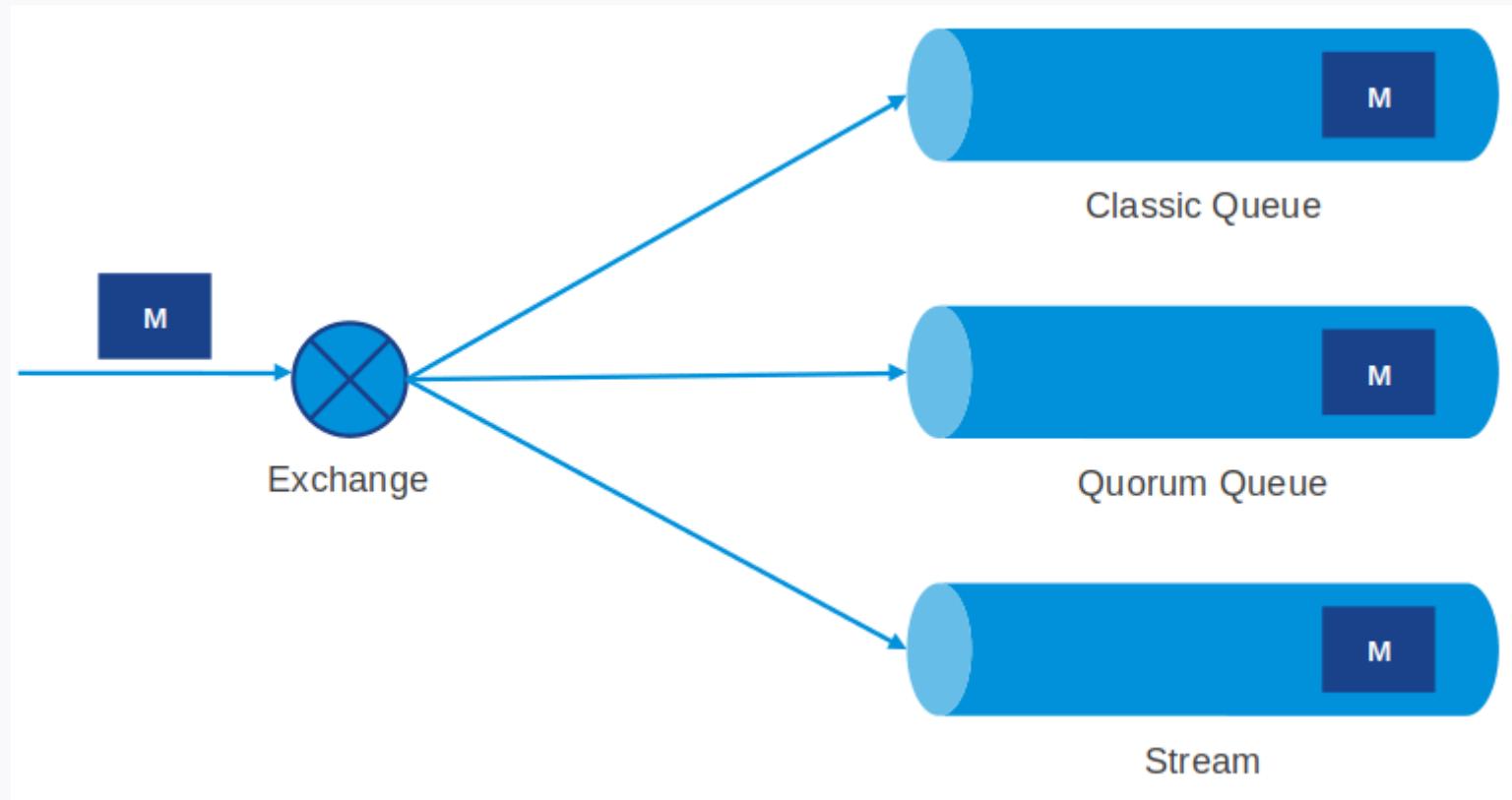


СОВМЕСТИМОСТЬ ПОТОКОВ

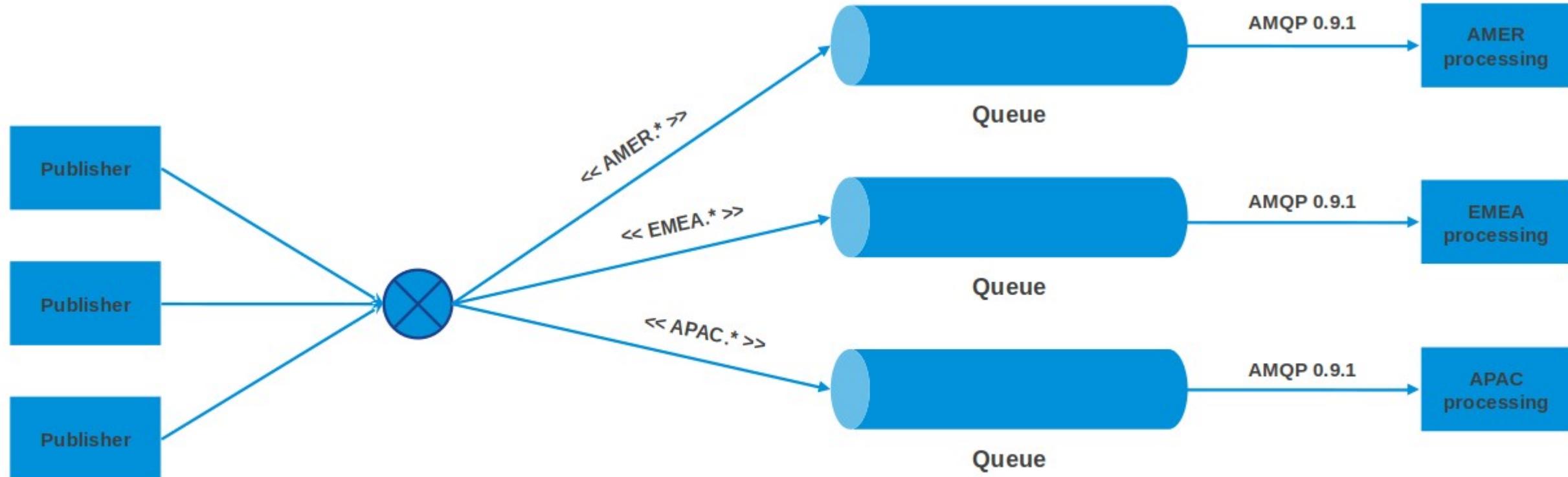
Поток доступен по протоколам AMQP 0.9.1, MQTT, STOMP и Stream



Поток отображается как обычная очередь AMQP 0.9.1

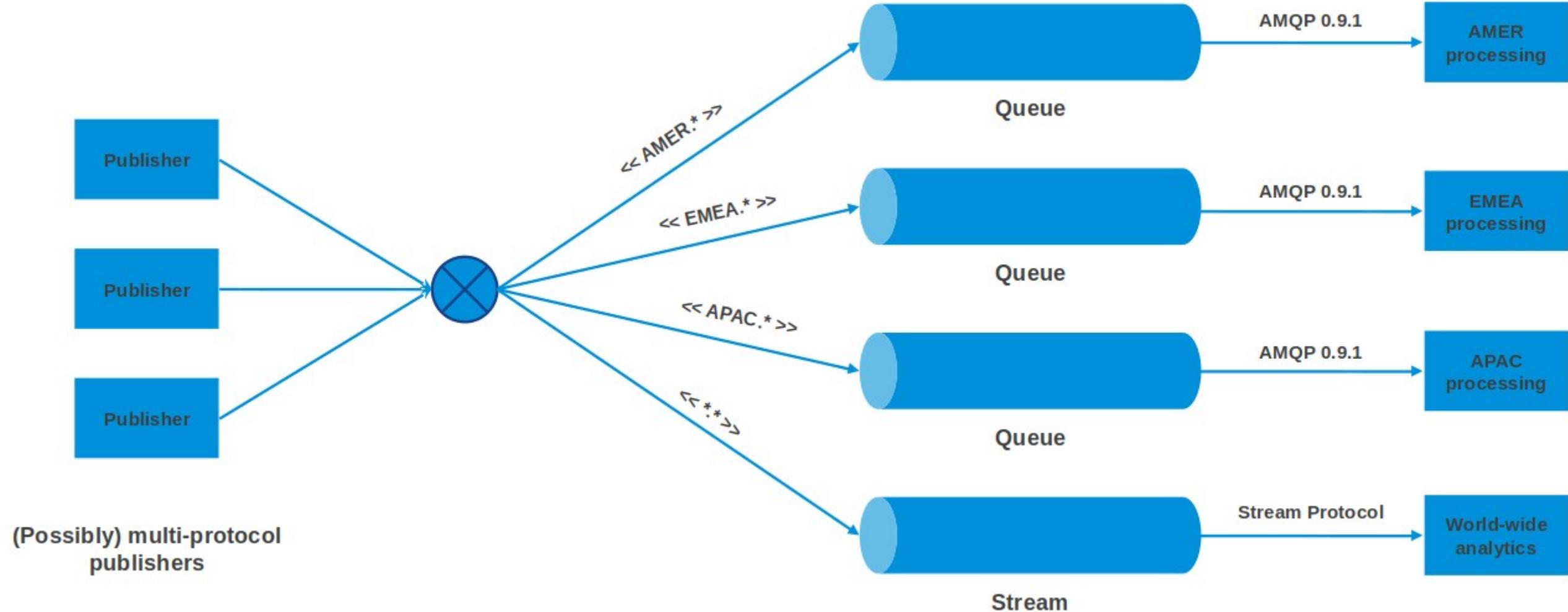


Традиционная маршрутизация с AMQP



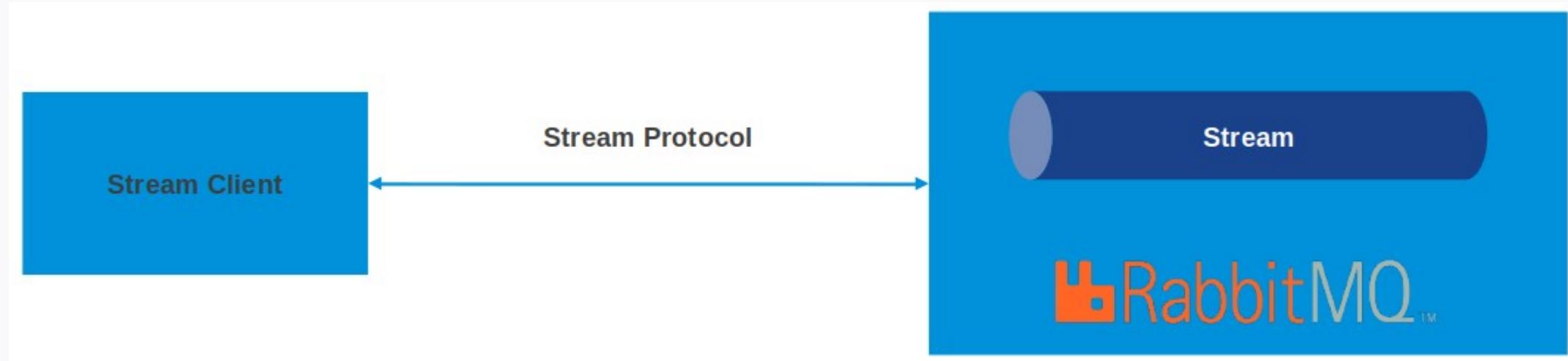
(Possibly) multi-protocol
publishers

Добавим поток для аналитики



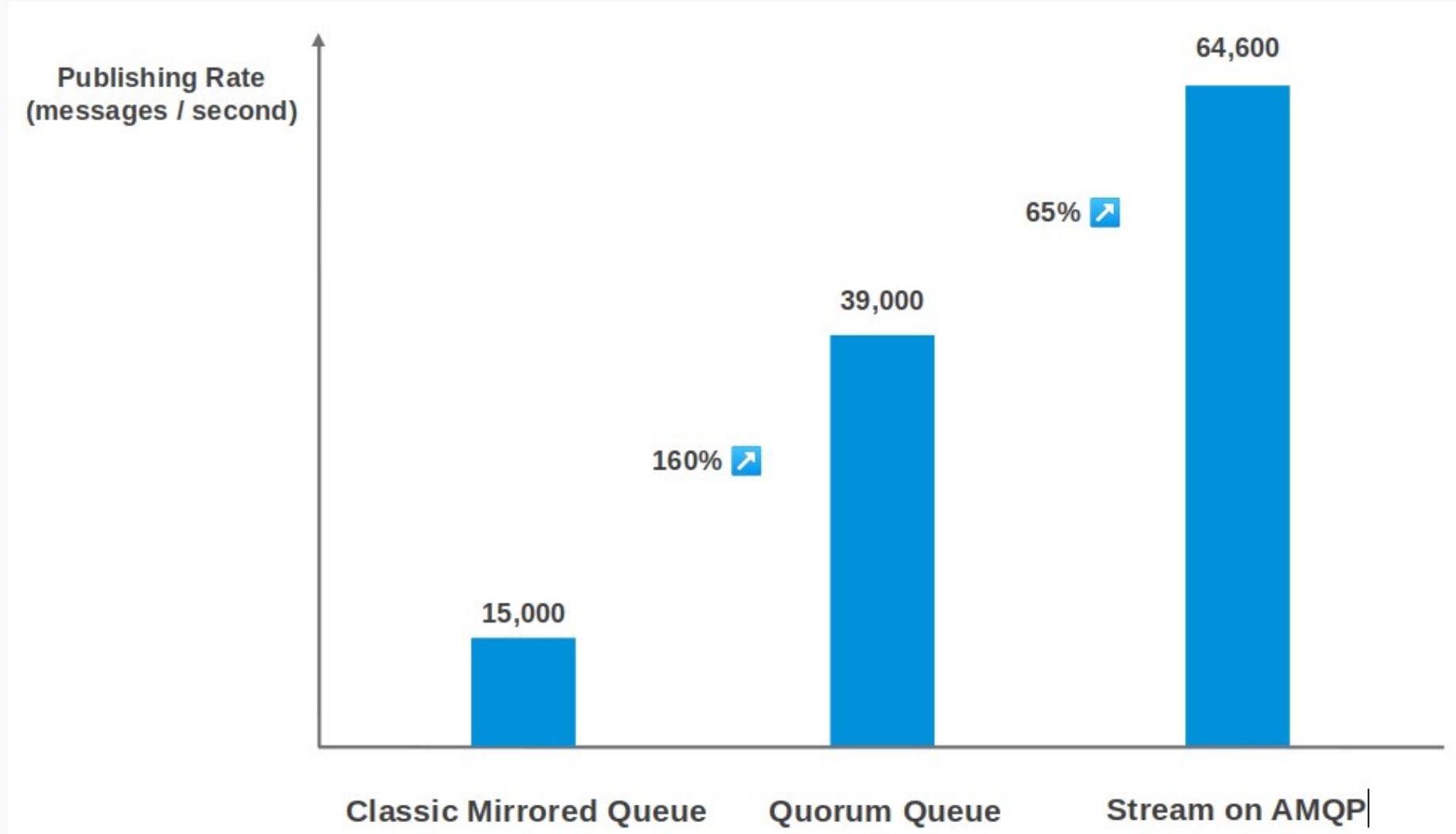
Прямое взаимодействие между клиентом и потоками

Exchange не участвует



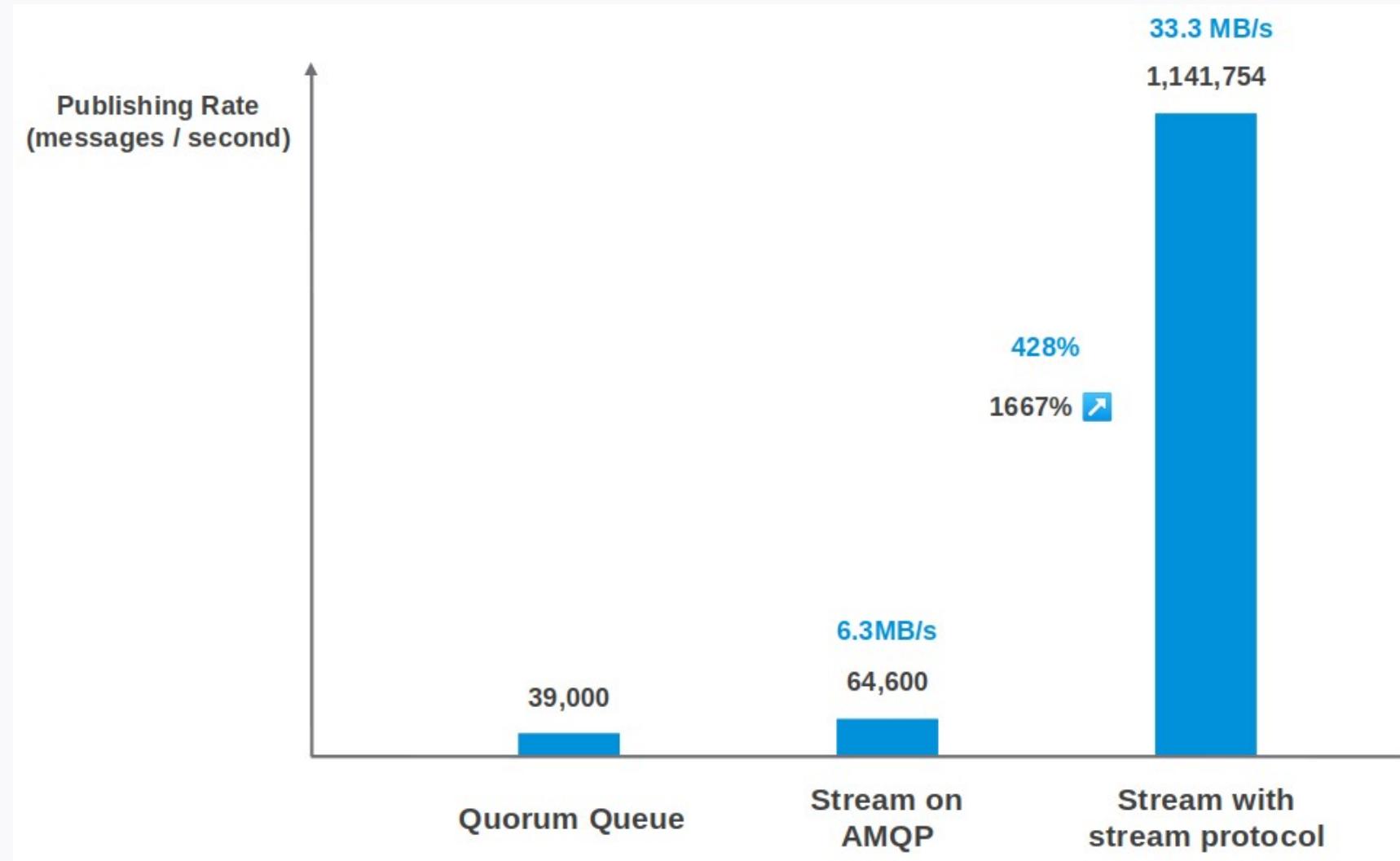
Производительность потоков на AMQP

3-node cluster (c2-standard-16 instances)



Производительность потоков на протоколе Stream

3-node cluster (c2-standard-16 instances)



Практика

- Запускаем RabbitMQ
 - ```
docker run -d --rm --name rabbitmq --network host -e RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS='"-rabbitmq_stream advertised_host localhost'" rabbitmq:management
```
- Включаем Stream Plugin
  - ```
docker exec rabbitmq rabbitmq-plugins enable rabbitmq_stream
```
- Запускаем тест
 - ```
docker run -it --rm --network host pivotalrabbitmq/stream-perf-test --uris rabbitmq-stream://localhost:5552
```



# RabbitMQ vs Apache Kafka

# RabbitMQ vs Apache Kafka

|                                      | <b>RabbitMQ</b>                                                                                                                                                                                                                                       | <b>Kafka</b>                                                                                                                         |
|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>Data structures</i>               | Message queues, logs                                                                                                                                                                                                                                  | Logs                                                                                                                                 |
| <i>Variety of messaging patterns</i> | Point-to-point, non-durable, pub-sub with routability, persistent log. Retry, dead-letter-exchange are possible.                                                                                                                                      | Pub-sub                                                                                                                              |
| <i>Interoperability</i>              | RabbitMQ natively supports multiple protocols, including MQTT, STOMP, JMS, AMQP, HTTP.<br><br>The features of Streams can be used via any AMQP client, and Streams can be connected to existing exchanges. There are multiple Stream clients planned. | Kafka supports its own protocol and HTTP.<br><br>There are multiple clients available.                                               |
| <i>Ease of operation</i>             | RabbitMQ is easy to get started with and is suitable for both small use cases (Raspberry PI, local dev's machine) all the way up to large scale enterprise use cases                                                                                  | Operationalizing Apache Kafka at scale is difficult. Confluent Platform can help however this cannot be deployed for every use case. |
| <i>Event Stream Processing</i>       | RabbitMQ relies on external components to perform this, such as Spring Cloud Streams                                                                                                                                                                  | Kafka uses Kafka Streams, which can then perform stateful transformations using separate components and state stores.                |

# Резюме

# Резюме

- **Система организации потоков** – система нежёсткого реального времени, которая делает данные доступными, когда клиентское приложение хочет их видеть.
- Очереди сообщений "разъединяют" отправителей и потребителей
- **RabbitMQ** и **Kafka** представляют разные подходы к организации очередей сообщений
- Надёжную доставку сообщений надо организовывать

# Литература

# Ссылки

- RabbitMQ – <https://www.rabbitmq.com>
- Tanzu RabbitMQ – <https://rabbitmq.com/tanzu>  
<https://tanzu.vmware.com/rabbitmq>
- RabbitMQ Streams Overview  
<https://blog.rabbitmq.com/posts/2021/07/rabbitmq-streams-overview>
- RabbitMQ Streams <https://www.rabbitmq.com/streams.html>
- RabbitMQ Stream Java Client  
<https://github.com/rabbitmq/rabbitmq-stream-java-client>
- Apache Kafka – <http://kafka.apache.org/documentation>
- Confluent Platform – <https://www.confluent.io/product/confluent-platform>
- Quick Start for Confluent Platform –  
<https://docs.confluent.io/platform/current/quickstart/ce-docker-quickstart.html>
- ksqlDB – <https://docs.ksqldb.io/en/latest>

# Список литературы

- Apache Kafka. Потоковая обработка и анализ данных –  
[https://www.piter.com/product\\_by\\_id/112863410](https://www.piter.com/product_by_id/112863410)
- Kafka Streams в действии –  
[https://www.piter.com/product\\_by\\_id/132649774](https://www.piter.com/product_by_id/132649774)
- Потоковая обработка данных –  
<https://dmkpress.com/catalog/computer/data/978-5-97060-606-3/>
- Проектирование событийно-ориентированных систем –  
<https://dmkpress.com/catalog/computer/os/978-5-6042412-1-9/>
- Распределенные системы  
<https://dmkpress.com/catalog/computer/programming/algorithms/978-5-97060-708-4/>

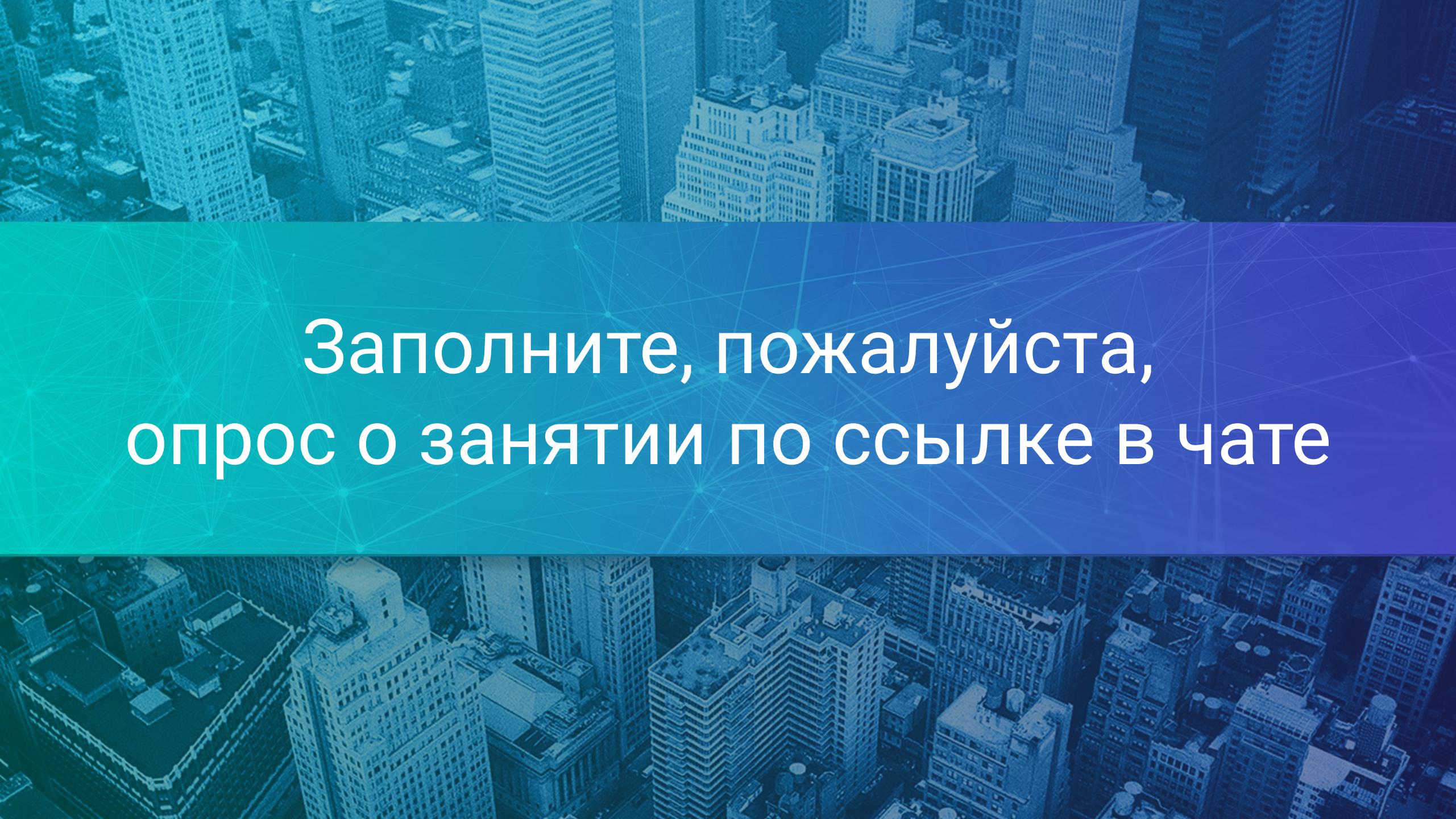
# Рефлексия



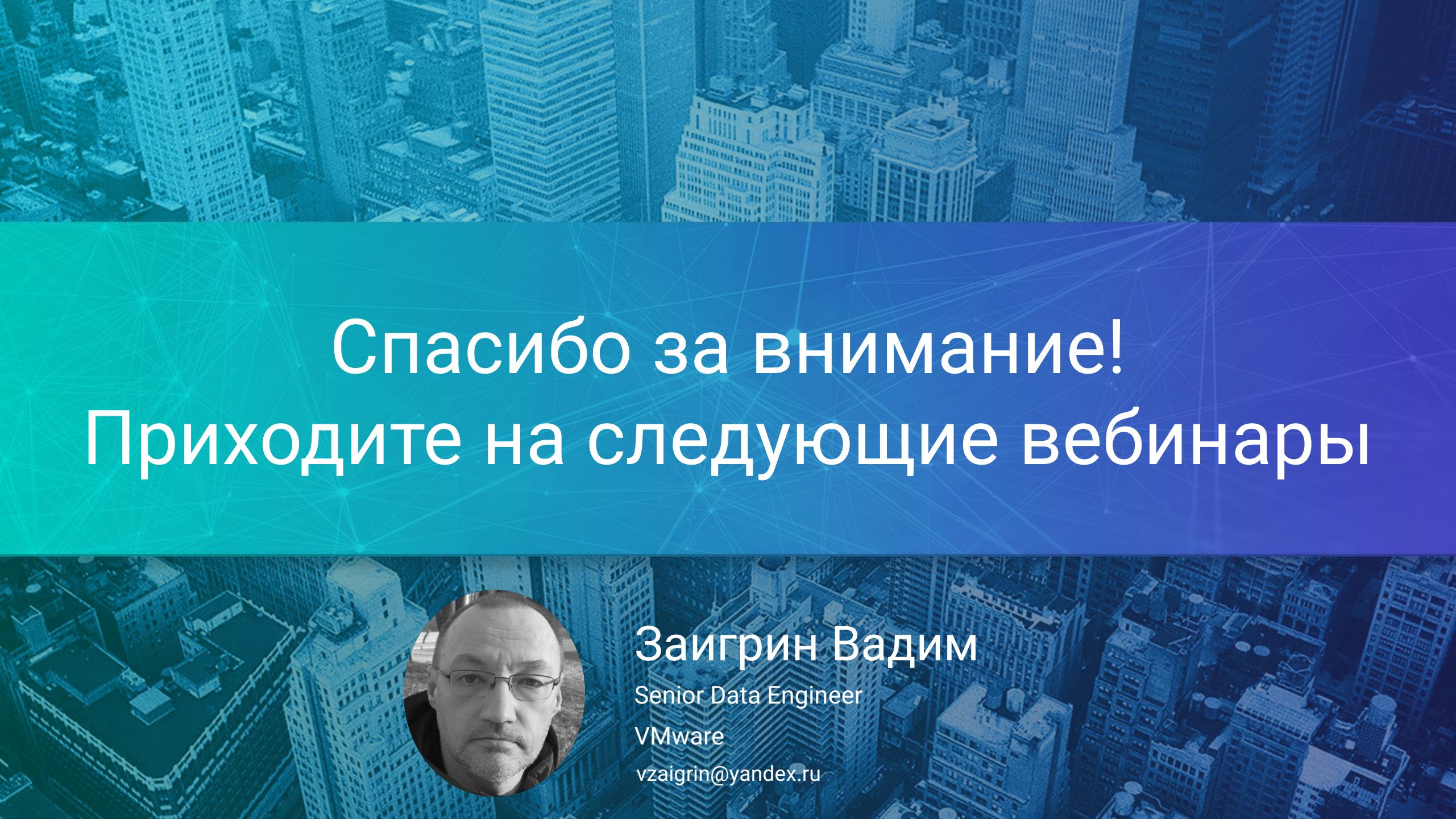
С какими основными мыслями уходите с вебинара?



Достигли ли вы цели вебинара?



Заполните, пожалуйста,  
опрос о занятии по ссылке в чате



Спасибо за внимание!  
Приходите на следующие вебинары



Заигрин Вадим

Senior Data Engineer

VMware

vzaigrin@yandex.ru