Шпаргалка
по Kafka

Когда мы начинаем изучать микросервисы , нам представляют множество различных концепций, шаблонов, протоколов и инструментов, таких как **Messaging , AMQP , RabbitMQ , Event-sourcing , gRPC , CQRS** и многие другие.

Среди всего этого есть и **Apache Kafka.** Но проблема в том, что люди часто неправильно понимают, что такое Kafka на самом деле, и порой можно увидеть, как они говорят о ней так, как будто это просто еще одна система обмена сообщениями. На самом деле, вы можете это сделать, но использование Kafka только для этого может оказаться пустой тратой ресурсов.

Команда FAANG School собрала для вас эту шпаргалку, чтобы объяснить, как работает Kafka, когда, как и почему вам стоит использовать ее в своих проектах. Подробно расскажем об основных концепциях, архитектуре, сценариях использованиях и еще множестве других, полезных вещей!

Что такое Apache Kafka?

Если вы зайдете на сайт Кафки, то прямо на первой странице вы найдете определение:

«Распределенная потоковая платформа»

Что такое «распределенная потоковая платформа»? Во-первых, нам нужно определить, что такое поток. Потоки — это просто бесконечные данные. Они просто продолжают прибывать из одного места в другое, и вы можете обрабатывать их в режиме реального времени.

А что тогда значит «распределенный» поток? «Распределенный» означает, что Kafka работает не на одном компьютере, а одновременно на нескольких компах, объединенных в группу — кластер. Каждый узел (отдельный компьютер) в кластере называется Broker. Эти брокеры — просто серверы, обычные железные компы с запущенной на них копией программы Apache Kafka.

Итак, по сути, **Kafka — это набор машин, работающих вместе, чтобы иметь возможность обрабатывать бесконечные данные в реальном времени.**

Ее распределенная архитектура — одна из причин, по которой Kafka так популярна. Брокеры (т.е. отдельные серверы с запущенной копией Kafka) — это то, что делает ее такой устойчивой, надежной, масштабируемой и отказоустойчивой. Вот почему Kafka так производительна и безопасна.

Но почему существует это заблуждение, что Kafka — это просто еще одна система обмена сообщениями? Чтобы ответить на этот вопрос, нам нужно сначала объяснить, как работает обмен сообщениями.

Обмен сообщениями

Обмен сообщениями, если говорить очень кратко, это просто действие по отправке сообщения из одной программы в другую. В нем есть три основных участника:

- **Производитель:** Программа, что создает и отправляет сообщения в одну или несколько очередей сообщений;

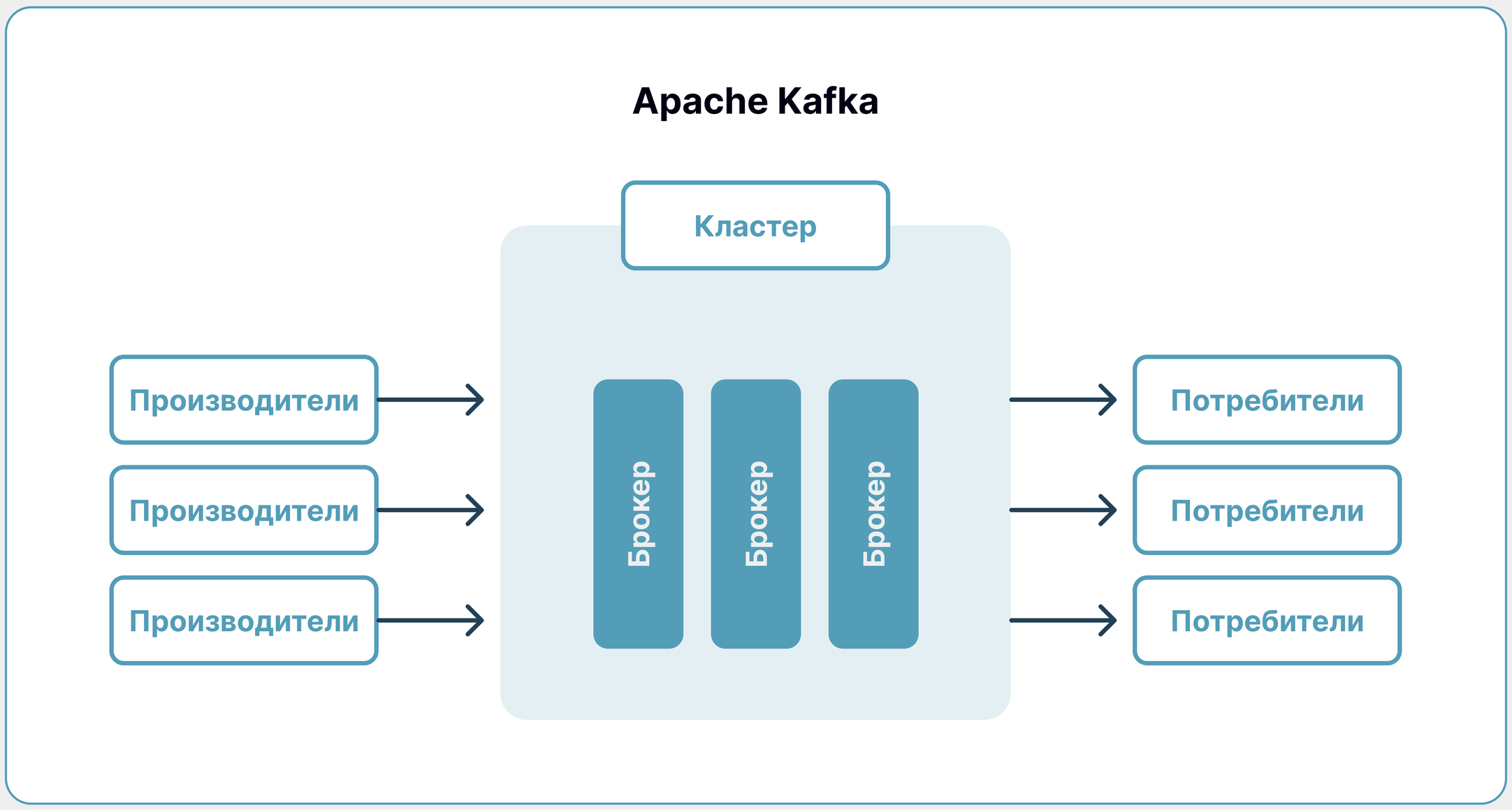
- **Очередь:** буферная структура данных, которая получает (от производителей) и доставляет сообщения (потребителям) в порядке FIFO (First-In First-Out). Когда сообщение доставлено, оно навсегда удаляется из очереди, и возможности получить его обратно нет;

- **Потребитель:** Программа, что подписана на одну или несколько очередей и получает их сообщения после публикации.

Вот и все, вот как работает обмен сообщениями. Как видите, здесь нет ничего о потоках, реальном времени, кластере (в зависимости от выбранного инструмента, вы также можете использовать кластер, но он не является нативным, как в Kafka).

Архитектура Кафки

Теперь, когда мы знаем, как работает обмен сообщениями, давайте погрузимся в мир Кафки. В Кафке у нас также есть **Производители** и **Потребители (Producer & Consumer)**, они работают очень похожим образом, как и в обмене сообщениями, как производя, так и потребляя сообщения.



Как видите, это очень похоже на то, что мы обсуждали при обмене сообщениями, но здесь у нас нет концепции **очереди**, вместо нее есть концепция **тем (topic)**.

Тема — это очень специфический тип потока данных, он очень похож на очередь, он также получает и доставляет сообщения, но есть некоторые концепции, которые нам необходимо понимать относительно тем :

- Тема делится на **разделы (partition)**, каждая тема может иметь один или несколько разделов, и нам нужно указать это число при создании темы.
  Вы можете представить тему как папку в операционной системе, а каждую папку внутри нее как раздел;

- Если мы не дадим никакого ключа сообщению при его создании, то по умолчанию производители будут отправлять сообщения по **кругу**: каждый раздел получит сообщение (даже если они отправлены одним и тем же производителем). Из-за этого мы не можем гарантировать порядок доставки на уровне раздела, если мы хотим всегда отправлять сообщение в один и тот же раздел, нам нужно дать ключ **нашим** сообщениям. Это гарантирует, что сообщение всегда будет отправлено в один и тот же раздел**; **
- Каждое сообщение будет сохранено на диске брокера и получит **смещение** (уникальный идентификатор — offset). Это смещение уникально на уровне раздела, каждый раздел имеет свои собственные смещения. Это еще одна причина, по которой Kafka такая особенная — она сохраняет сообщения на диске (как база данных, и на самом деле Kafka тоже является базой данных), чтобы иметь возможность восстановить их позже, если это необходимо. В отличие от обычной системы обмена сообщениями, где сообщение удаляется после использования;
- Производители используют смещение (offset) для чтения сообщений, они читают от самого старого к самому новому. В случае отказа потребителя, когда он восстановится, то начнет читать с последнего смещения;

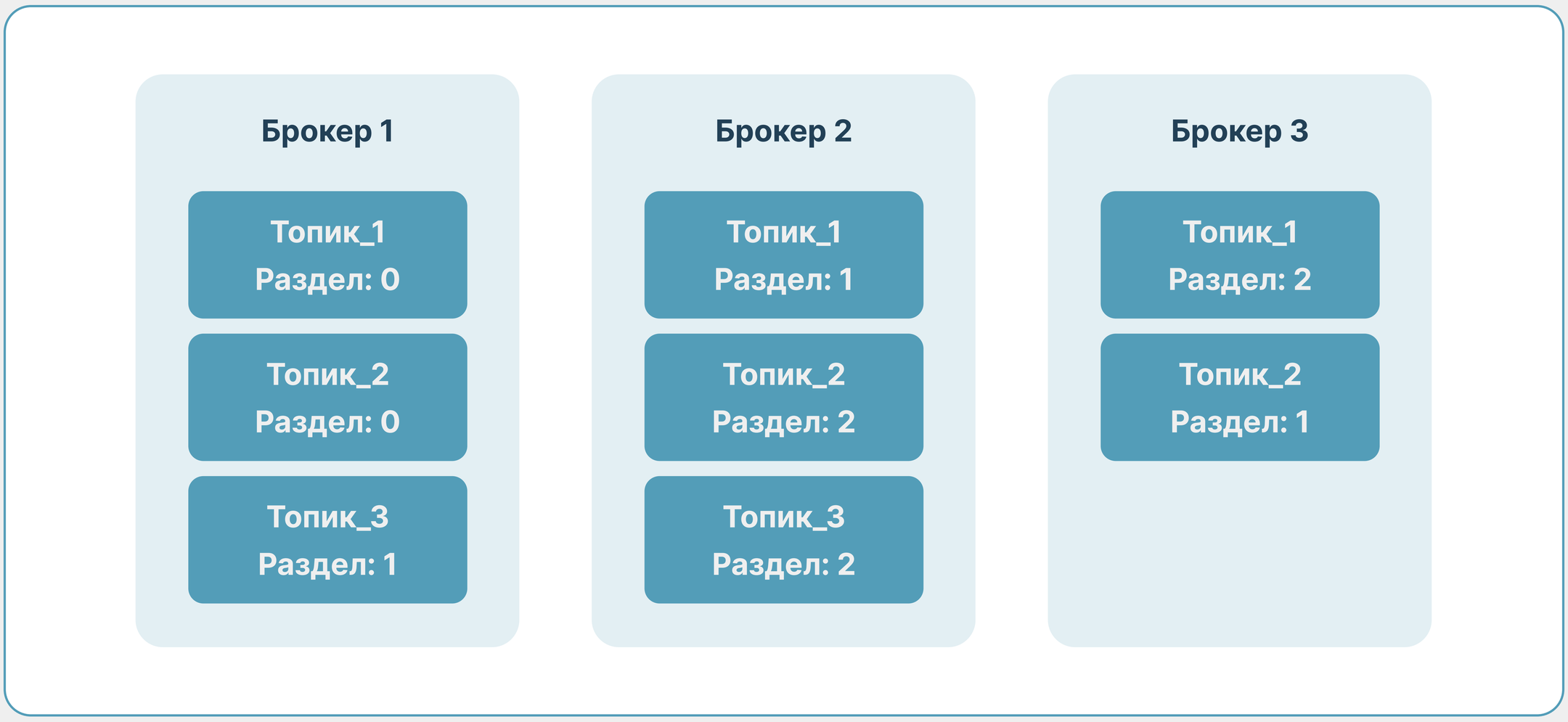
Разметка разделов топика Kafka

Брокеры

Как уже было сказано, Kafka работает распределенным образом. Кластер Kafka может содержать столько брокеров, сколько нужно по мере необходимости.

Каждый брокер в кластере имеет уникальный идентификатор и содержит по крайней мере один раздел темы (топика). Чтобы настроить количество копий разделов в каждом брокере, нам нужно настроить то, что называется **Replication Factor** при создании темы.

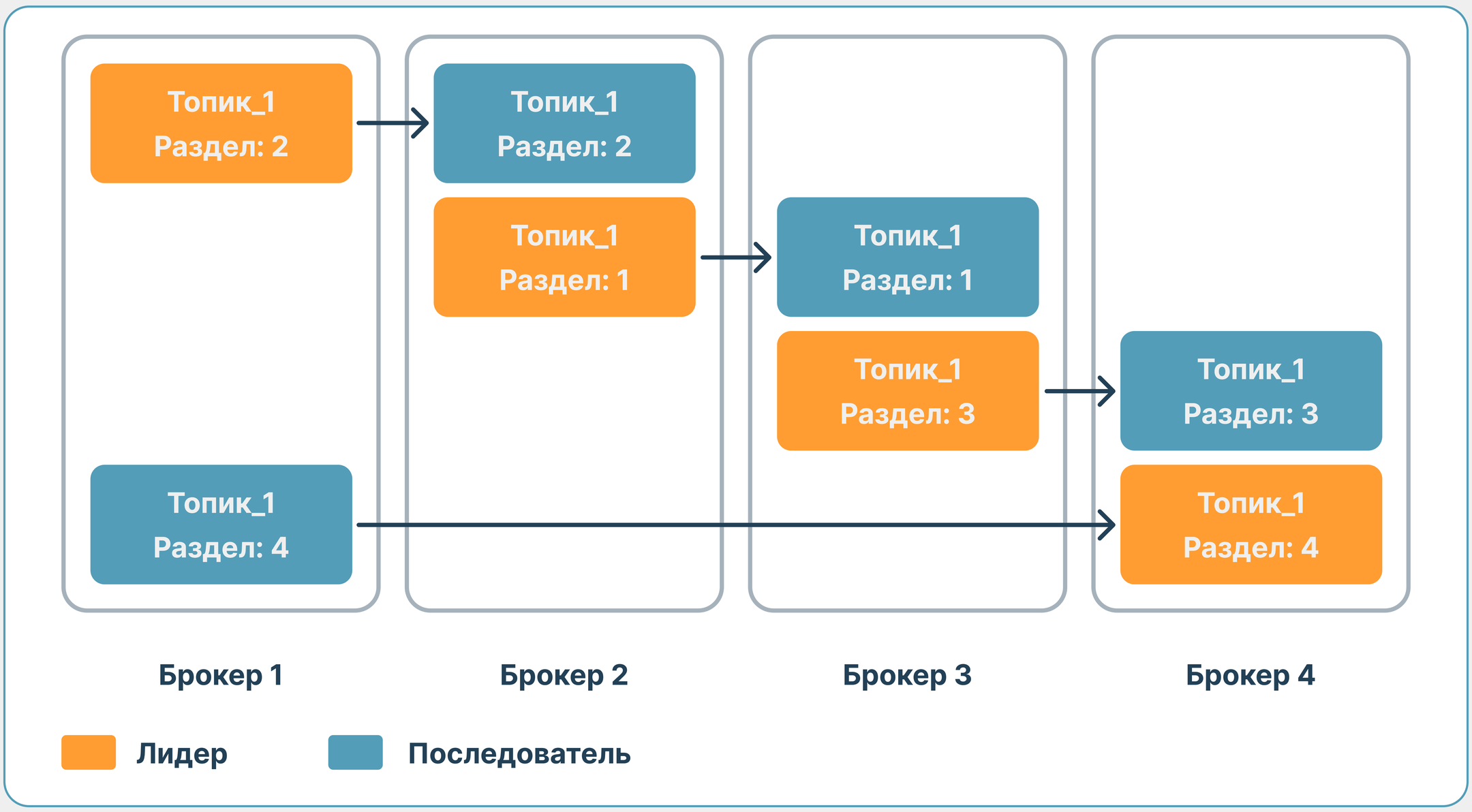
Допустим, у нас есть три брокера в нашем кластере, тема с тремя разделами и Replication Factor, равным трем. В этом случае каждый брокер будет отвечать за один раздел темы, а также каждый раздел еще и будет скопирован на каждый брокер.



Очень важно, чтобы количество разделов соответствовало количеству брокеров, таким образом, каждый брокер будет отвечать за один раздел темы.

Чтобы обеспечить надежность кластера, Kafka вводит концепцию **Partition Leader.** Каждый раздел темы в брокере является лидером раздела и может существовать только один лидер на раздел. Лидер — единственный, кто получает сообщения, его **реплики** будут просто синхронизировать данные (они должны быть синхронизированы с этим). Это гарантирует, что даже если брокер выйдет из строя, его данные не будут потеряны из-за реплик.

Когда лидер выходит из строя, Zookeeper автоматически выбирает его копию в качестве нового лидера.



На изображении выше **Брокер 1** является лидером **Раздела 1** Темы **1** и имеет реплику в **Брокере 2**. Предположим, что **Брокер 1** умирает (компьютер, на котором он запущен, сгорает или выключается). Когда это происходит, Zookeeper обнаруживает это изменение и делает **Брокера 2** лидером **Раздела 1**. Именно это делает распределенную архитектуру Kafka такой мощной.

Производители

Как и в мире обмена сообщениями, продюсеры в Kafka — это программы, которые производят и отправляют сообщения в темы (топики).

Как уже было сказано, сообщения отправляются по **кругу**. Пример: сообщение 01 отправляется в раздел 0 темы 1, а сообщение 02 — в раздел 1 той же темы. Это означает, что мы не можем гарантировать, что сообщения, созданные одним и тем же производителем, всегда будут доставлены в один и тот же раздел (partition). Нам нужно указать ключ при отправке сообщения, если мы хотим, чтобы все сообщения с таким ключом оказались в одном и том же разделе. Сообщения, находящиеся в одном и том же разделе сохраняют порядок следования, тогда как сообщения в разных разделах могут быть потреблены в разном порядке.

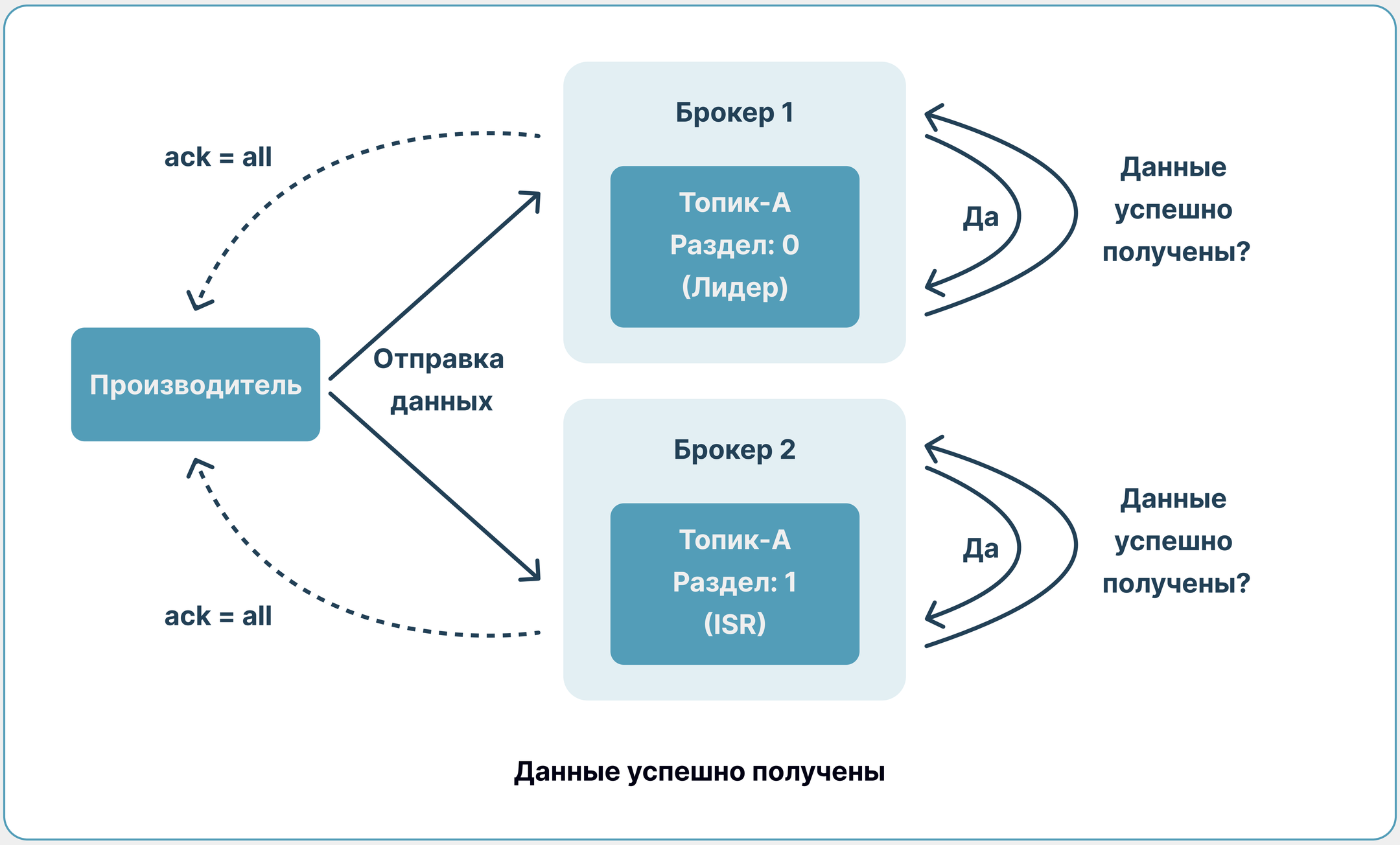
Соответственно, когда мы понимаем, что нам важно сохранить порядок следования сообщений, то добавляем один и тот же ключ к тем, которые мы хотим отправить в один и тот же раздел. Kafka сгенерирует хэш на основе этого ключа и будет знать, в какой раздел доставить это сообщение.

Этот хеш учитывает количество разделов темы, поэтому это количество нельзя изменить, когда тема уже создана.

ACK

Когда мы работаем с концепцией сообщений, есть нечто, называемое **Подтверждением (ack)**. Подтверждение — это, по сути, информация о том, что сообщение было доставлено в брокер. В Kafka мы можем настроить тот, как такие подтверждения будут отправлены производителю сообщений и будут ли вообще. Для этого есть три разных уровня конфигурации:

- **ack = 0:** здесь мы говорим, что не хотим получать подтверждений от Kafka. В случае сбоя брокера сообщение будет потеряно. Тем не менее скорость работы Kafka гораздо выше при этой настройке;
- **ack = 1:** Это конфигурация по умолчанию, с помощью которой мы говорим, что хотим получить подтверждение от лидера раздела. Данные будут потеряны только в случае выхода из строя лидера раздела (все еще есть шанс);
- **ack = all:** Это самая надежная конфигурация. Мы говорим, что хотим получать подтверждение не только от лидера, но и от его реплик. Т.е. сообщение гарантированно окажется и в разделе на брокере — лидере, а также совершенно точно будет скопировано и на другие машины, где расположены копии этого раздела. Это самая безопасная конфигурация, поскольку потеря данных исключена. Тем не менее это самая медленная настройка, т. к. Kafka будет ждать синхронизации всех машин между друг другом при копировании сообщения между ними.



Данные успешно получены

Потребители и потребительские группы

Потребители — это приложения, подписанные на одну или несколько тем, которые будут читать сообщения оттуда. Они могут читать из одного или нескольких разделов.

Когда потребитель читает только из одного раздела, мы можем гарантировать порядок чтения, но когда один потребитель читает из двух или более разделов, он будет читать параллельно, поэтому нет никакой гарантии порядка чтения. Например, сообщение, пришедшее позже, может быть прочитано раньше другого, пришедшего раньше.

Вот почему нам нужно быть осторожными при выборе количества разделов и при создании сообщений.

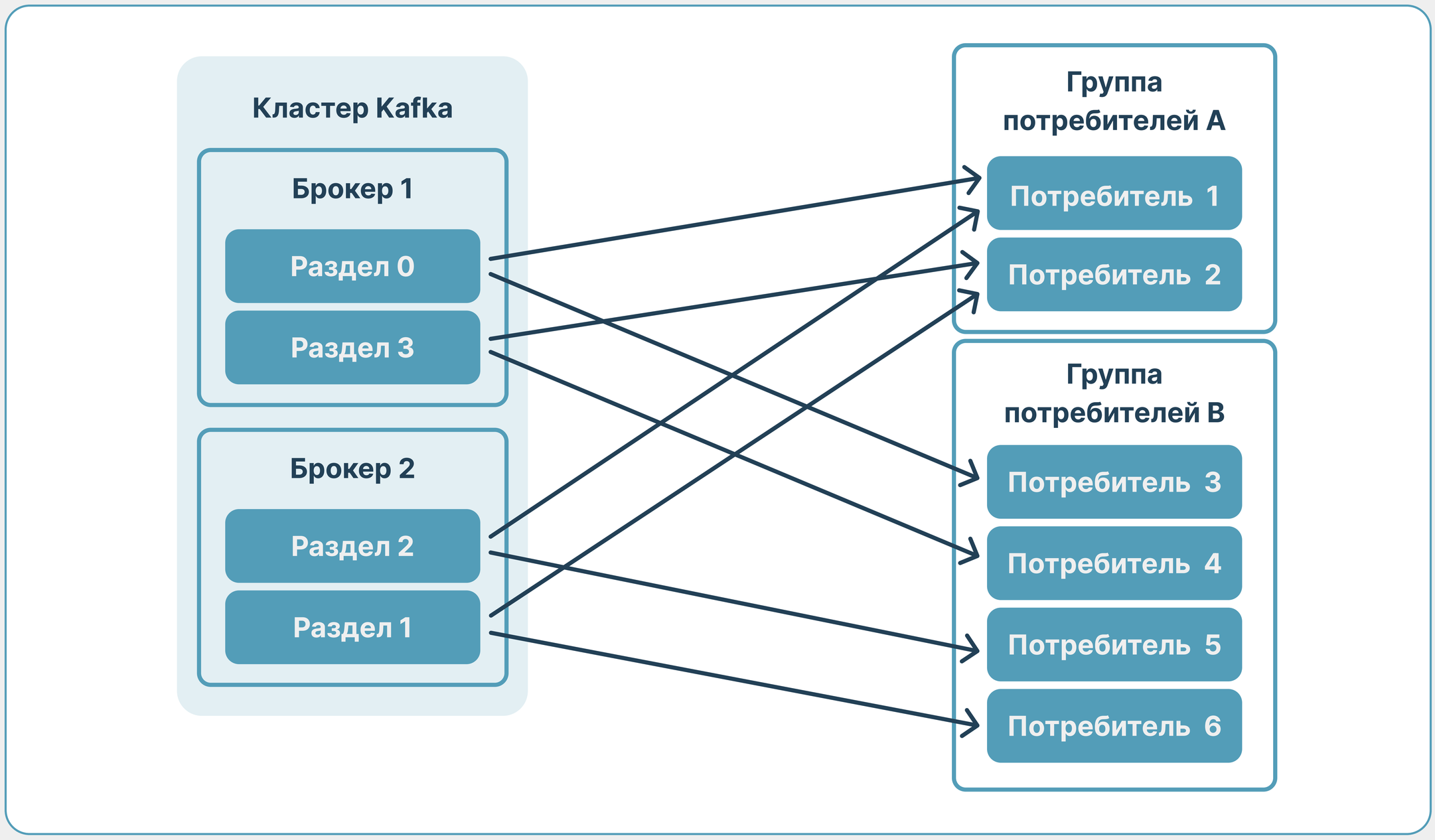
Еще одна важная концепция Kafka — это Consumer Groups (потребительские группы). Это действительно важно, когда нам нужно масштабировать чтение сообщений.

Это становится очень затратным, когда одному потребителю необходимо считывать данные из многих разделов, поэтому нам необходимо распределить нагрузку между нашими потребителями, и вот тут-то вступают в дело группы потребителей.

Данные из одной темы будут распределены между потребителями, благодаря чему мы можем гарантировать, что наши потребители смогут обрабатывать большие объемы данных.

Идеал — иметь в группе такое же количество потребителей, сколько у нас разделов в теме. Таким образом, каждый потребитель читает только из одного раздела — наиболее эффективный сценарий.

Но при добавлении потребителей в группу нужно быть осторожным: если количество потребителей станет больше количества разделов, некоторые потребители не будут читать ни из одного раздела и останутся бездействующими, а то пустая трата ресурсов и денег.



Обратите внимание, что Kafka позиционируется как платформа потоковой передачи событий, поэтому термин «сообщение», который часто используется в очередях сообщений, в Kafka не используется. Мы называем это «событием».

Топ-5 сценариев использования Kafka

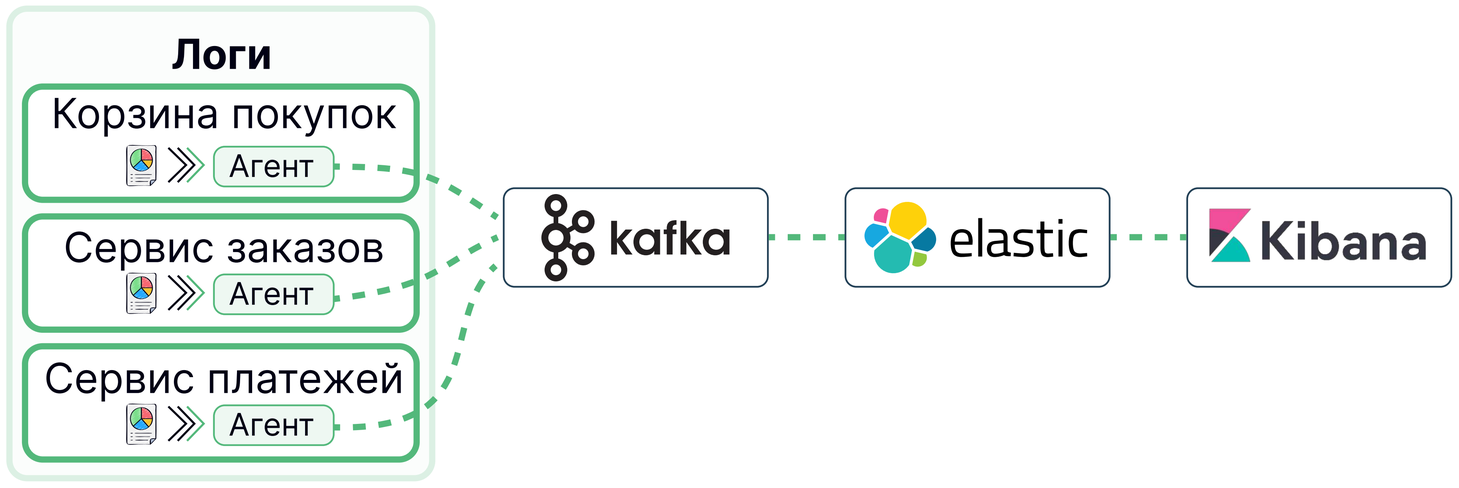
Kafka изначально была разработана для массовой обработки логов. Она сохраняет сообщения до истечения срока их жизни и позволяет потребителям забирать сообщения в своем темпе.

Давайте рассмотрим популярные сценарии использования Kafka.

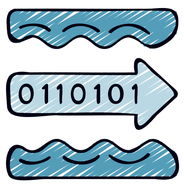
1. Обработка и анализ логов
2. Потоковая передача данных в системе рекомендаций
3. Мониторинг систем и оповещение
4. CDC (отслеживание изменений данных)
5. Миграция систем

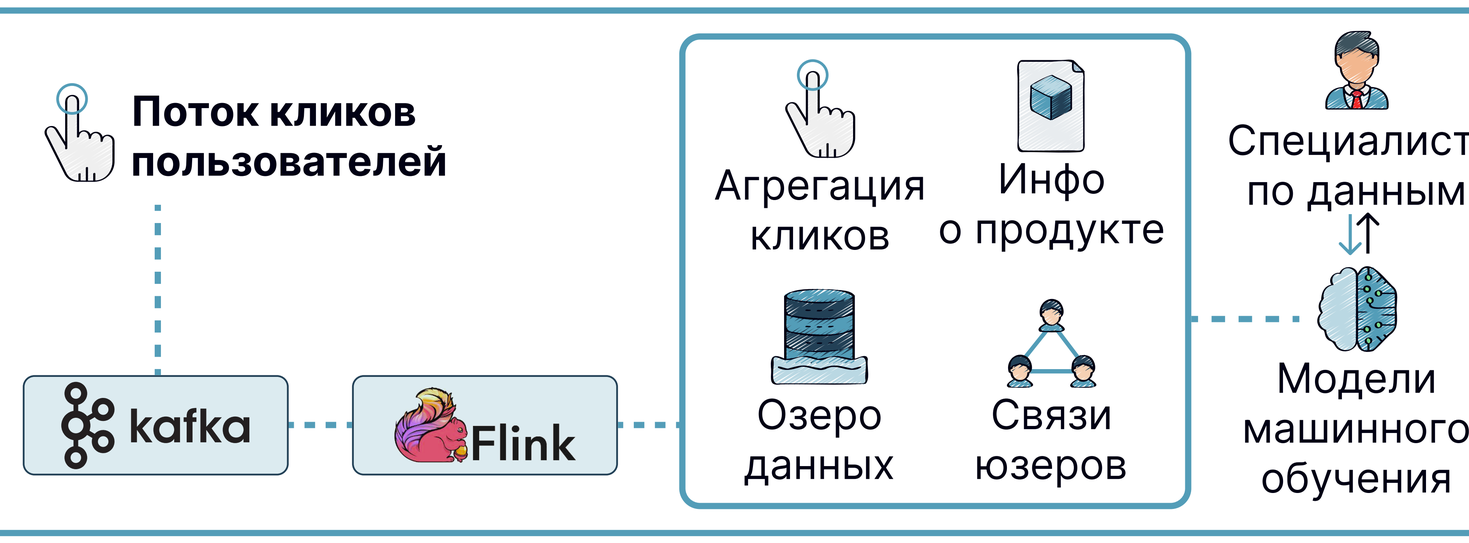
Топ 5 сценариев использования Kafka

Анализ логов

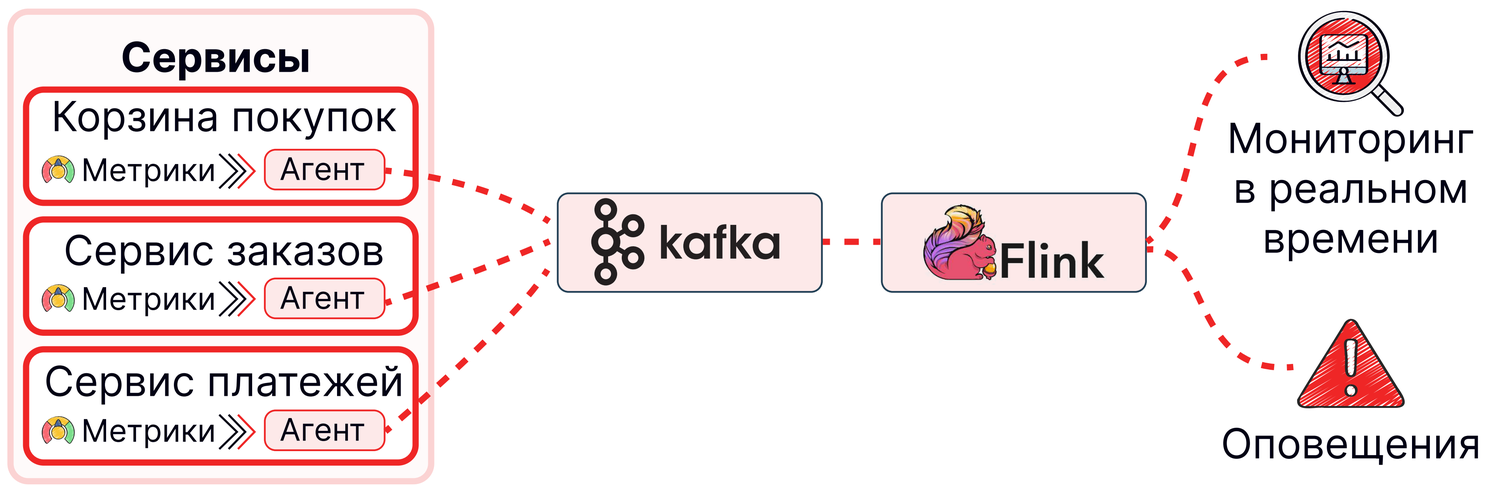


Передача данных в рекомендациях

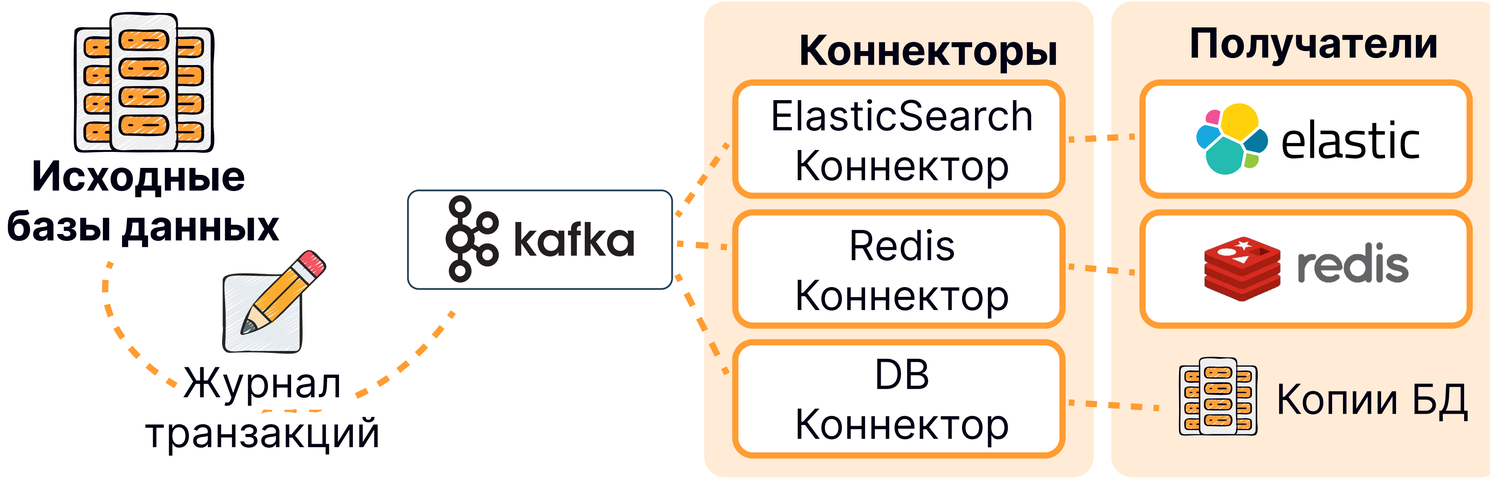




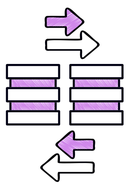
Системный мониторинг и оповещение

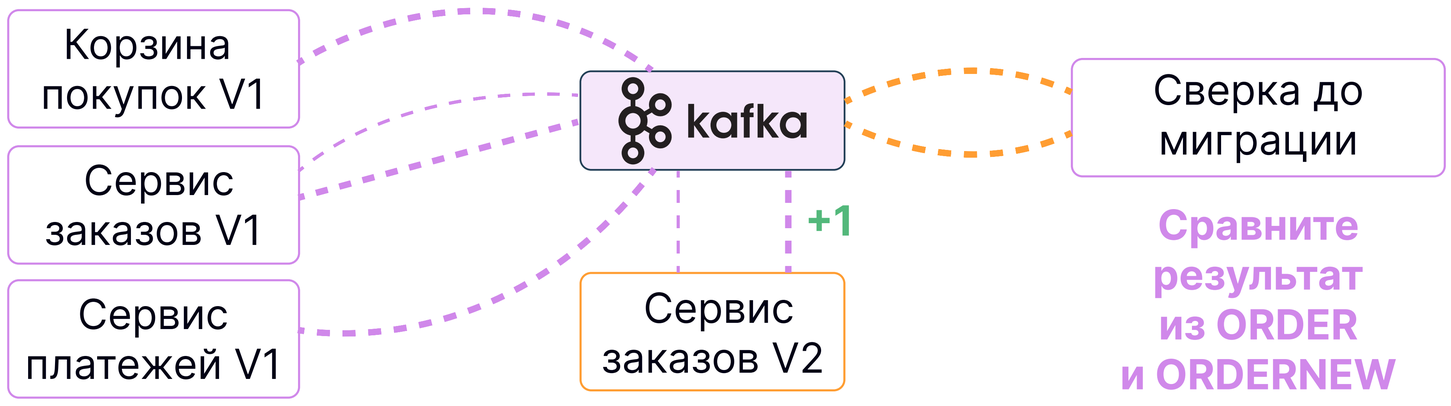


Захват изменений данных



Миграция системы





Как выбрать очередь сообщений? Kafka против RabbitMQ

В настоящее время Kafka является продуктом, к которому обращаются, когда нужно использовать очередь сообщений в проекте. Однако это не всегда лучший выбор, если мы рассматриваем конкретные требования.

Очередь, поддерживаемая SQL БД

Давайте рассмотрим пример работы Starbucks для продажи кофе. Два самых важных требования:

- Асинхронная обработка, позволяющая кассиру принять следующий заказ без ожидания варки кофе для предыдущего.
- Надежность хранения данных, чтобы заказы клиентов не были упущены в случае возникновения проблем.

Порядок сообщений здесь не имеет большого значения, поскольку кофеварки часто делают партии одного и того же напитка. Масштабируемость также не так важна, поскольку очереди ограничены каждой точкой Starbucks.

Очереди Starbucks можно реализовать в таблице обычной SQL базы данных вместо Kafka. На схеме ниже показано, как это работает:

Последовательность заказов	Статус заказов	Обновление по времени
100	NEW	09:00:03
101	IN_PROGRESS	09:00:25
102	IN_PROGRESS	09:01:02
103	DONE	09:01:41

Когда кассир принимает заказ, в очереди, поддерживаемой базой данных, создается новый заказ. Затем кассир может принять другой заказ, пока кофеварка забирает новые заказы партиями. После завершения заказа кофеварка отмечает его как выполненный в базе данных. Затем клиент забирает свой кофе на стойке.

Работа по очистке таблицы от больше не нужных данных может запускаться в конце каждого дня для удаления выполненных заказов (то есть тех, которые имеют статус «ВЫПОЛНЕНО»).

Для варианта использования Starbucks простая очередь в базе данных соответствует требованиям без необходимости использования Kafka. Таблица заказов с операциями CRUD (Create-Read-Update-Delete) работает отлично.

Зачем делать так вместо использования Kafka? Это может быть банально проще. Скорее всего у вас в приложении уже есть SQL база данных. Порой вместо затаскивания в систему еще одного компонента — Kafka — ее настройки и поддержки гораздо проще использовать компонент, который уже доступен в системе. Это часто быстрее и дешевле, если, конечно, это в самом деле решает вашу проблему.

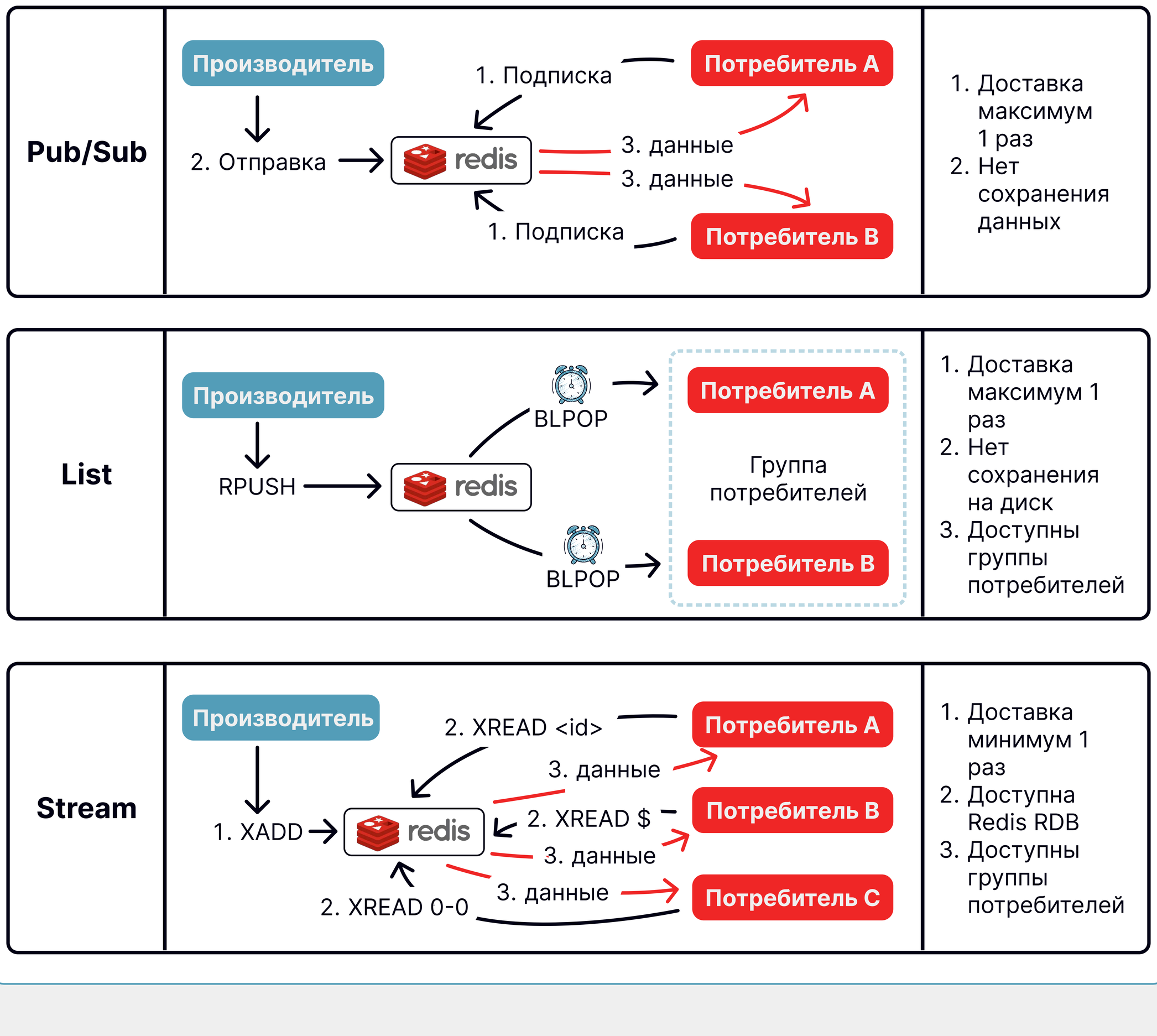
Очередь, поддерживаемая Redis

Очередь сообщений на основе базы данных все еще требует разработки для создания таблицы очереди и чтения/записи из нее. Для небольшого стартапа с ограниченным бюджетом, который уже использует Redis для кэширования, Redis также может служить очередью сообщений.

Существует 3 способа использования Redis в качестве очереди сообщений:

1. Паб/Саб (pub/sub)
2. Список
3. Поток

На схеме ниже показано, как они работают.



Pub/Sub удобен, но имеет некоторые ограничения по доставке. Потребитель подписывается на ключ и получает данные, когда производитель публикует данные по тому же ключу. Ограничение заключается в том, что данные доставляются не более одного раза (at most once доставка).

Если потребитель вышел из строя и не получил опубликованные данные, эти данные теряются. Кроме того, данные не сохраняются на диске. Если Redis выходит из строя, все данные Pub/Sub теряются. Pub/Sub подходит для мониторинга метрик, где допустима некоторая потеря данных.

Структура данных List в Redis может создавать очередь FIFO (First-In-First-Out). Потребитель использует BLPOP для ожидания сообщений в режиме блокировки, поэтому следует применять тайм-аут. Потребители, ожидающие в одном и том же List, образуют группу потребителей, где каждое сообщение потребляется только одним потребителем. Как структура данных Redis, List может быть сохранен на диске.

Stream решает ограничения двух вышеперечисленных методов. Потребители выбирают, откуда читать сообщения — «$» для новых сообщений, «<id>» для определенного идентификатора сообщения или «0-0» для чтения с самого начала.

Подводя итог, можно сказать, что очереди сообщений, поддерживаемые базой данных и Redis, легко поддерживать. Если они не могут удовлетворить ваши потребности, лучше подойдут специализированные продукты для очередей сообщений. Далее мы сравним два самых популярных варианта.

RabbitMQ против Kafka

Крупным компаниям, которым требуются надежные, масштабируемые и удобные в обслуживании системы, следует оценить доступные очереди сообщений по следующим параметрам:

- Функциональность
- Производительность
- Масштабируемость
- Экосистема

В таблице ниже сравниваются два типичных продукта очереди сообщений: RabbitMQ и Kafka.

	RabbitMQ	kafka
Написан на	Erlang	Scala, Java
Протокол	AMQP	Двоичный протокол поверх TCP
API клиента	Java, Ruby, JavaScript, Go, C, Swift, Spring, Elixir, PHP, and .NET	Java, Ruby, Python, Node.js
Гибкие правила маршрутизации	Поддержка в компоненте Exchange	Нет
Потребление сообщений	push	pull
Приоритет сообщений	✓	Нет
Упорядочивание сообщений	Упорядоченные в очереди	Упорядоченные в теме
Удаление сообщений	Удаление по ACK	Удаление по истечении периода хранения
Безопасность	Управление доступом через админские инструменты	TKS, JAAS
Масштабируемость	RabbitMQ consistent hash exchange	Добавление большего числа партиций к теме
Отказоустойчивость	✓	✓
Нагрузка от скопления сообщений	Плохо справляется	Предназначен для удержания сообщений
Производительность	Десятки тысяч/сек	Миллионы/сек
Экосистема	Не так хороша, как у Kafka	Хорошо поддерживается в области больших данных и потоковых вычислений

Как они работают

RabbitMQ работает как промежуточное ПО для обмена сообщениями — оно отправляет сообщения потребителям, а затем удаляет их после подтверждения. Это позволяет избежать накопления сообщений, которое RabbitMQ считает проблематичным.

Kafka изначально был создан для массовой обработки журналов. Он сохраняет сообщения до истечения срока действия и позволяет потребителям извлекать сообщения в своем собственном темпе.

Языки и API

RabbitMQ написан на Erlang, что делает изменение основного кода сложным. Однако он предлагает очень богатый клиентский API и поддержку библиотек.

Kafka использует Scala и Java, но также имеет клиентские библиотеки и API для популярных языков, таких как Python, Ruby и Node.js.

Производительность и масштабируемость

RabbitMQ написан на Erlang, что делает изменение основного кода сложным. Однако он предлагает очень богатый клиентский API и поддержку библиотек.

Kafka использует Scala и Java, но также имеет клиентские библиотеки и API для популярных языков, таких как Python, Ruby и Node.js.

Экосистема

Многие современные приложения для больших данных и потоковой передачи данных интегрируют Kafka по умолчанию. Это делает его естественным для этих вариантов использования.

Варианты использования очереди сообщений

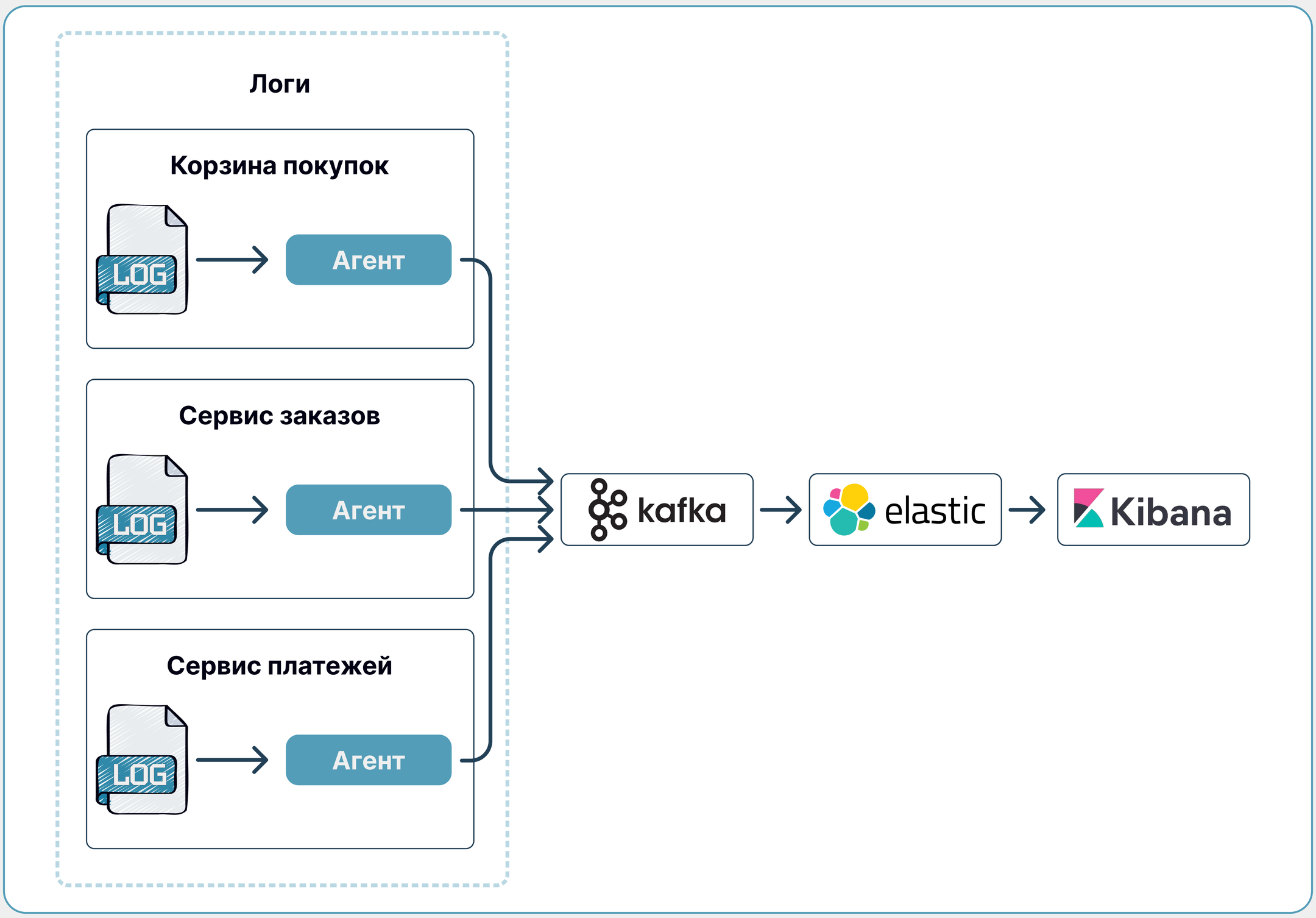
Теперь, когда мы рассмотрели особенности различных очередей сообщений, давайте рассмотрим несколько примеров того, как выбрать правильный продукт.

Обработка и анализ журналов

Для сайта электронной коммерции с такими услугами, как корзина покупок, заказы и платежи, нам необходимо анализировать журналы для изучения заказов клиентов.

На схеме ниже показана типичная архитектура, использующая стек «ELK»:

- ElasticSearch — индексирует журналы для полнотекстового поиска
- LogStash — агент сбора логов
- Kibana — пользовательский интерфейс для поиска и визуализации журналов
- Kafka — распределенная очередь сообщений



Углубленная тема: Архитектура Kafka. Почему она такая быстрая?

На картинке ниже собраны подробные сведения об архитектуре Kafka и структуре ее клиентского API. Мы видим, что хотя производитель, потребитель и брокер по-прежнему являются ключевыми элементами архитектуры, для создания кластера Kafka с высокой пропускной способностью и низкой задержкой требуется больше. Давайте рассмотрим компоненты по одному.

Вычислительный уровень

- Агрегация
- Фильтрация
- Группировка
- Объединение

Источник

Базы данных
Другие системы

Потребитель

Kafka Streams
KStreams
Ktable

Сбор данных

Базы данных
Поисковые системы
Другие системы

Приложение

REST/SQL/CLI

ksqlDB

Производитель

Kafka Connect

Kafka Connect

API Производителя

API Потребителя

Запись

временная метка
ключ
значение
заголовки

Пакет записей

Запись
Запись
Запись
сжатие

поток событий

Уровень хранения

Управляющий уровень

консенсус:
zookeeper (старый)
KRaft (новый)

Контроллер **Контроллер** **Контроллер**

Брокер 1

топик: заказы
раздел 0
0 1 2 3 4 ....

топик: платежи
раздел 1
0 1 2 3 4 ....

Брокер 2

топик: заказы
раздел 2
0 1 2 3 4 ....

топик: платежи
раздел 0
0 1 2 3 4 ....

Брокер 3

топик: заказы
раздел 1
0 1 2 3 4 ....

топик: заказы
раздел 3
0 1 2 3 4 ....

Уровень данных

обработка запросов клиентов

репликация данных

- неизменяемый
- только добавление
- смещение увеличивается монотонно

С точки зрения высокого уровня архитектура состоит из двух уровней: вычислительного уровня и уровня хранения.

Вычислительный слой

Вычислительный уровень, или уровень обработки, позволяет различным приложениям взаимодействовать с брокерами Kafka через API.

Производители используют API производителя. Если внешние системы, такие как базы данных, хотят общаться с Kafka, он также предоставляет Kafka Connect в качестве API интеграции.

Потребители общаются с брокером через API потребителя. Чтобы направить события в другие приемники данных, такие как поисковая система или база данных, мы можем использовать API Kafka Connect. Кроме того, потребители могут выполнять потоковую обработку с API Kafka Streams.

Если мы имеем дело с неограниченным потоком записей, мы можем создать KStream. Фрагмент кода ниже создает KStream для темы «заказы» с Serdes (сериализаторами и десериализаторами) для ключа и значения. Если нам просто нужен последний статус из журнала изменений, мы можем создать KTable для поддержания статуса. Kafka Streams позволяет нам выполнять агрегацию, фильтрацию, группировку и объединение потоков событий.

```
final KStreamBuilder builder = new KStreamBuilder();
final KStream<String, OrderEvent> orderEvents = builder
                    .stream(Serdes.String(), orderEventSerde, "orders");
```

Хотя API Kafka Streams отлично работает для Java приложений, иногда нам может понадобиться развернуть чистое задание потоковой обработки без встраивания его в приложение. Тогда мы можем использовать ksqlDB, кластер базы данных, оптимизированный для потоковой обработки. Он также предоставляет нам REST API для запроса результатов.

Мы видим, что с различной поддержкой API в вычислительном слое можно довольно гибко объединять операции, которые мы хотим выполнять в потоках событий. Например, мы можем подписаться на тему «orders», агрегировать заказы на основе продуктов и отправлять количество заказов обратно в Kafka в теме «ordersByProduct», на которую может подписаться и отобразить другое аналитическое приложение.

Уровень хранения

Этот слой состоит из брокеров Kafka. Брокеры Kafka работают на кластере серверов. Данные хранятся в разделах в разных темах. Тема похожа на таблицу базы данных, а разделы в теме могут быть распределены по узлам кластера. Внутри раздела события строго упорядочены по их смещениям. Смещение представляет собой положение события в разделе и увеличивается монотонно. События, сохраняемые на брокерах, являются неизменяемыми и могут быть только добавлены, даже удаление моделируется как событие удаления. Таким образом, производители обрабатывают только последовательные записи, а потребители — только последовательное чтение.

Обязанности брокера Kafka включают управление разделами, обработку чтения и записи, а также управление репликациями разделов. Он разработан так, чтобы быть простым и, следовательно, легко масштабируемым. Мы рассмотрим архитектуру брокера более подробно.

Поскольку брокеры Kafka развернуты в кластерном режиме, для управления узлами необходимы два компонента: плоскость управления и плоскость данных.

Плоскость управления

Плоскость управления управляет метаданными кластера Kafka. Раньше Zookeeper управлял контроллерами: в качестве контроллера выбирался один брокер. Теперь Kafka использует новый модуль KRaft для реализации плоскости управления. Несколько брокеров выбираются в качестве контроллеров.

Почему Zookeeper был исключен из зависимости кластера? С Zookeeper нам нужно поддерживать два отдельных типа систем: один — Zookeeper, а другой — Kafka. С KRaft нам нужно поддерживать только один тип системы, что значительно упрощает настройку и развертывание. Кроме того, KRaft более эффективно распространяет метаданные брокерам.

Мы не будем обсуждать здесь детали консенсуса KRaft. Следует помнить, что кэши метаданных в контроллерах и брокерах синхронизируются через специальную тему в Kafka.

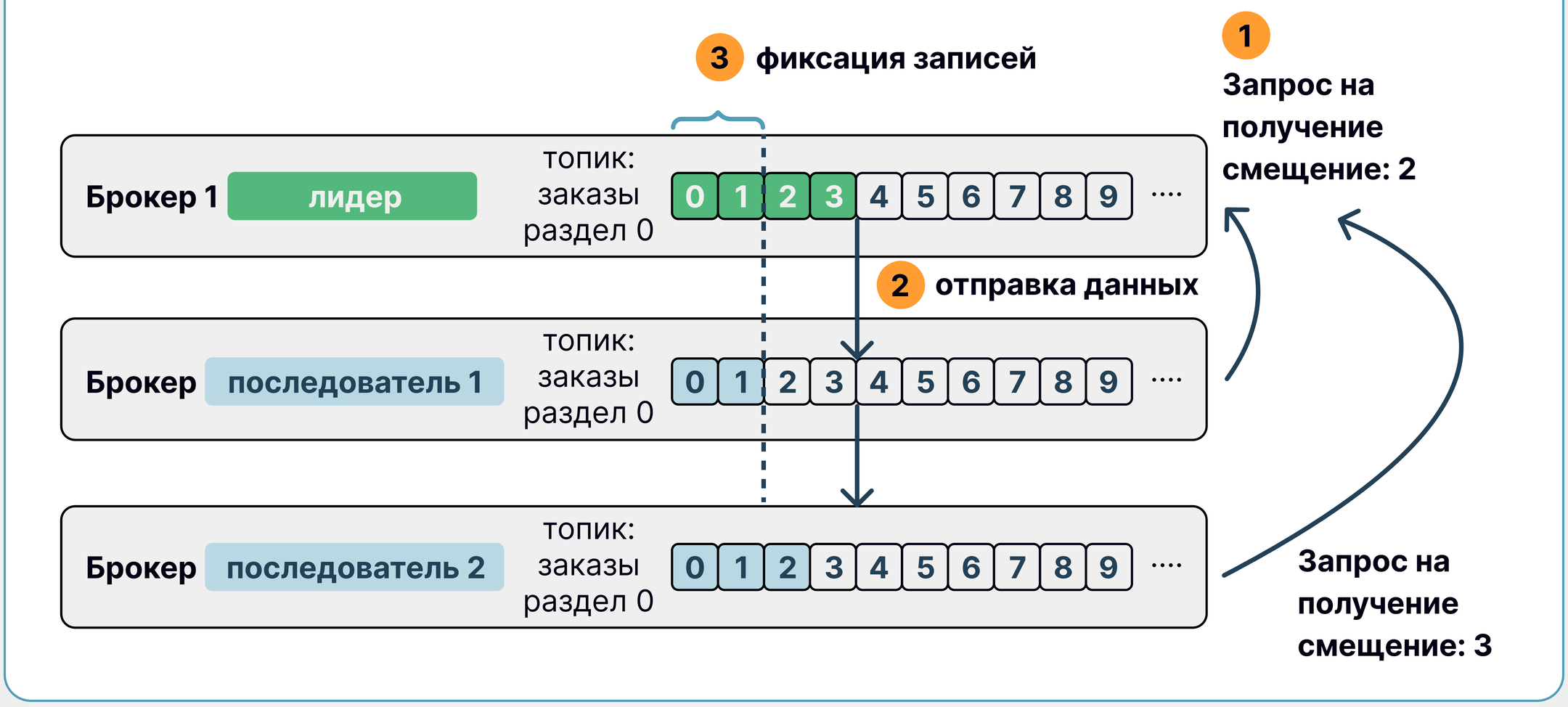
Плоскость данных

Плоскость данных обрабатывает репликацию данных. На схеме ниже показан пример. Раздел 0 в теме «orders» имеет 3 реплики на 3 брокерах. Раздел на брокере 1 является лидером, где текущее смещение данных равно 4; разделы на брокере 2 и 3 являются последователями, где смещения равны 2 и 3.

Шаг 1 — Чтобы догнать лидера, Последователь 1 отправляет запрос FetchRequest со смещением 2, а Последователь 2 отправляет запрос FetchRequest со смещением 3.

Шаг 2 — Затем лидер отправляет данные двум последователям соответственно.

Шаг 3 — Поскольку запросы последователей неявно подтверждают получение ранее извлеченных записей, лидер фиксирует записи до смещения 2.



Запись

Kafka использует класс Record как абстракцию события. Неограниченный поток событий состоит из множества Records.

В записи есть 4 части:

1. Временная метка
2. Ключ
3. Значение
4. Заголовки (необязательно)

Ключ используется для обеспечения порядка, совместного размещения данных, имеющих тот же ключ, и сохранения данных. Ключ и значение представляют собой массивы байтов, которые могут быть закодированы и декодированы с использованием сериализаторов и десериализаторов (serdes).

Детали работы брокера

Мы обсудили брокеры как уровень хранения. Данные организованы в темы и хранятся в виде разделов на брокерах. Теперь давайте рассмотрим, как работает брокер в деталях.

Шаг 1: Производитель отправляет запрос брокеру, который сначала попадает в приемный буфер сокета брокера.

Шаги 2 и 3: Один из сетевых потоков забирает запрос из буфера приема сокета и помещает его в общую очередь запросов. Поток привязан к конкретному клиенту-производителю.

Шаг 4: Пул потоков ввода-вывода Kafka выбирает запрос из очереди запросов.

Шаги 5 и 6: Поток ввода-вывода проверяет CRC данных и добавляет их в журнал фиксации. Журнал фиксации организован на диске в сегменты. В каждом сегменте есть две части: фактические данные и индекс.

Шаг 7: Запросы производителя сохраняются в структуре-чистилище для репликации, чтобы поток ввода-вывода мог быть освобожден для обработки следующего запроса.

Шаг 8: После репликации запроса он удаляется из чистилища. Генерируется ответ и помещается в очередь ответов.

Шаги 9 и 10: Сетевой поток забирает ответ из очереди ответов и отправляет его в соответствующий буфер отправки сокета. Обратите внимание, что сетевой поток привязан к определенному клиенту. Только после отправки ответа на запрос сетевой поток примет другой запрос от конкретного клиента.