# Оглавление

ГЛАВА 1

# Знакомьтесь: Kafka

Деятельность любого предприятия основана на данных. Мы получаем информацию, анализируем ее, выполняем над ней какие-либо действия и создаем новые данные в качестве результатов. Все приложения создают данные — сообщения журнала, показатели, информацию об операциях пользователей, исходящие сообщения или что-то еще. Каждый байт данных что-нибудь да значит — что-нибудь, определяющее дальнейшие действия. А чтобы понять, что именно, нам нужно переместить данные из места создания туда, где их можно проанализировать. Это мы каждый день наблюдаем на таких сайтах, как Amazon, где щелчки кнопкой мыши на интересующих нас товарах превращаются в предлагаемые нам же чуть позже рекомендации.

От скорости этого процесса зависят адаптивность и оперативность нашего предприятия. Чем меньше усилий мы тратим на перемещение данных, тем больше можем уделить внимания основной деятельности. Именно поэтому конвейер — ключевой компонент на ориентированном на работу с данными предприятии. Способ перемещения данных оказывается практически так же важен, как и сами данные.

> Первопричиной всякого спора ученых является нехватка данных. Постепенно мы приходим к согласию относительно того, какие данные нужны, получаем эти данные, и данные решают проблему. Или я оказываюсь прав, или вы, или мы оба ошибаемся. И можно двигаться дальше.
>
> *Нил Деграсс Тайсон (Neil deGrasse Tyson)*

## Обмен сообщениями по типу «публикация/подписка»

Прежде чем перейти к обсуждению нюансов Apache Kafka, важно понять концепцию обмена сообщениями по типу «публикация/подписка» и причину, по которой она является критически важным компонентом приложений, управляемых данными. *Обмен сообщениями по типу «публикация/подписка»* (publish/subscribe (pub/sub) messaging) — паттерн проектирования, отличающийся тем,

что отправитель (издатель) элемента данных (сообщения) не направляет его конкретному потребителю. Вместо этого он каким-то образом классифицирует сообщения, а потребитель (подписчик) подписывается на определенные их классы. В системы типа «публикация/подписка» для упрощения этих действий часто включают брокер — центральный пункт публикации сообщений.

## С чего все начинается

Множество сценариев использования публикации/подписки начинается одинаково — с простой очереди сообщений или канала обмена ими между процессами. Например, вы создали приложение, которому необходимо отправлять куда-либо мониторинговую информацию, для чего приходится открывать прямое соединение между вашим приложением и приложением, отображающим показатели на инструментальной панели, и передавать последние через это соединение (рис. 1.1).


**Рис. 1.1.** Единый непосредственный издатель показателей

Это простое решение простой задачи, удобное для начала мониторинга. Но вскоре вам захочется анализировать показатели за больший период времени, а на инструментальной панели это не слишком удобно. Вы создадите новый сервис для получения показателей, их хранения и анализа. Для поддержки этого измените свое приложение так, чтобы оно могло записывать их в обе системы. К этому времени у вас появятся еще три генерирующих показатели приложения, каждое из которых будет точно так же подключаться к этим двум сервисам. Один из коллег предложит идею активных опросов сервисов для оповещения, так что вы добавите к каждому из приложений сервер, выдающий показатели по запросу. Вскоре у вас появятся дополнительные приложения, использующие эти серверы для получения отдельных показателей в различных целях. Архитектура станет напоминать рис. 1.2, возможно, соединениями, которые еще труднее отслеживать.

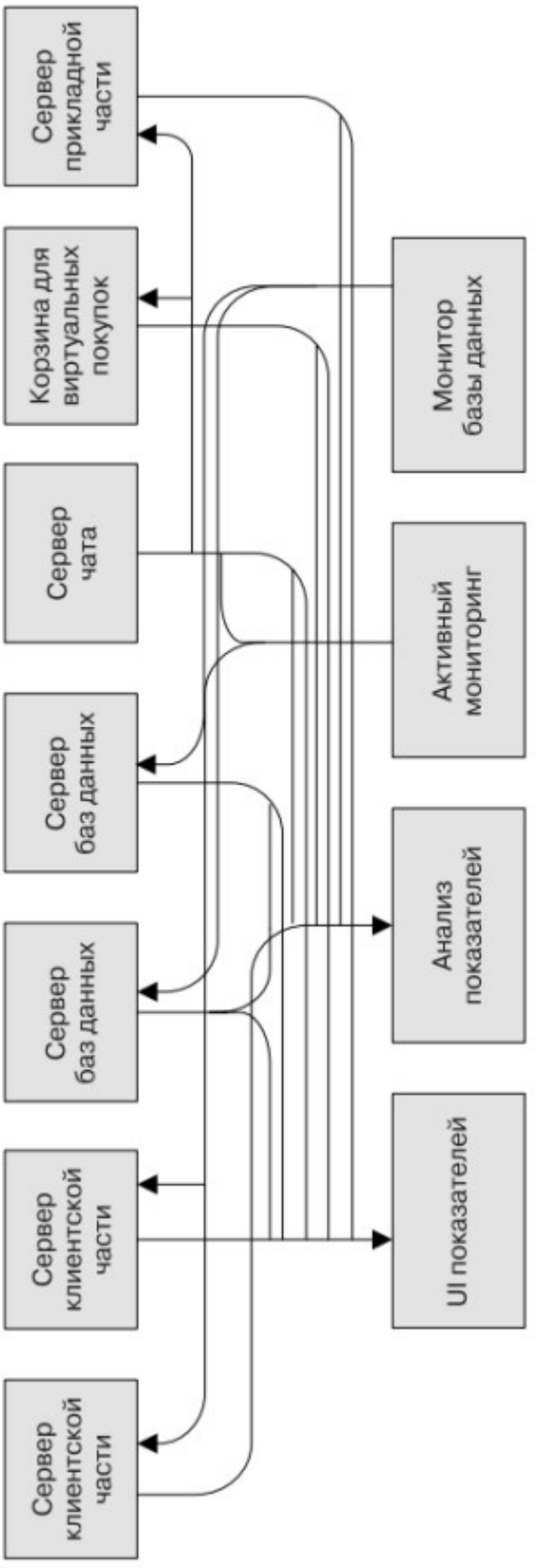


**Рис. 1.2.** Множество издателей показателей, использующих прямые соединения

Некоторая недоработка тут очевидна, так что вы решаете ее исправить. Создаете единое приложение, получающее показатели от всех имеющихся приложений и включающее сервер, — у него станут их запрашивать все системы, которым нужны эти показатели. Благодаря этому сложность архитектуры уменьшается (рис. 1.3). Поздравляем, вы создали систему обмена сообщениями по типу «публикация/подписка»!

## Отдельные системы организации очередей

В то время как вы боролись с показателями, один из ваших коллег аналогичным образом трудился над сообщениями журнала. А еще один работал над отслеживанием действий пользователей на веб-сайте клиентской части и передачей этой информации разработчикам, занимающимся машинным обучением, параллельно с формированием отчетов для начальства. Вы все шли одним и тем же путем, создавая системы, разделяющие издателей информации и подписчиков на нее. Инфраструктура с тремя отдельными системами публикации/подписки показана на рис. 1.4.

Использовать ее намного лучше, чем прямые соединения (см. рис. 1.2), но возникает существенное дублирование. Компании приходится сопровождать несколько систем организации очередей, в каждой из которых имеются собственные ошибки и ограничения. А между тем вы знаете, что скоро появятся новые сценарии обмена сообщениями. Необходима единая централизованная система, поддерживающая публикацию обобщенных типов данных, которая могла бы развиваться по мере расширения вашего бизнеса.

# Открываем для себя систему Kafka

Apache Kafka была разработана в качестве системы обмена сообщениями по принципу «публикация/подписка», предназначенной для решения описанной задачи. Ее часто называют распределенным журналом фиксации транзакций, а в последнее время — распределенной платформой потоковой обработки. Журнал фиксации файловой системы или базы данных предназначены для обеспечения долговременного хранения всех транзакций таким образом, чтобы можно было их воспроизвести с целью восстановления согласованного состояния системы. Аналогично данные в Kafka хранятся долго, упорядоченно, и их можно читать когда угодно. Кроме того, они могут распределяться по системе в качестве меры дополнительной защиты от сбоев, равно как и ради повышения производительности.

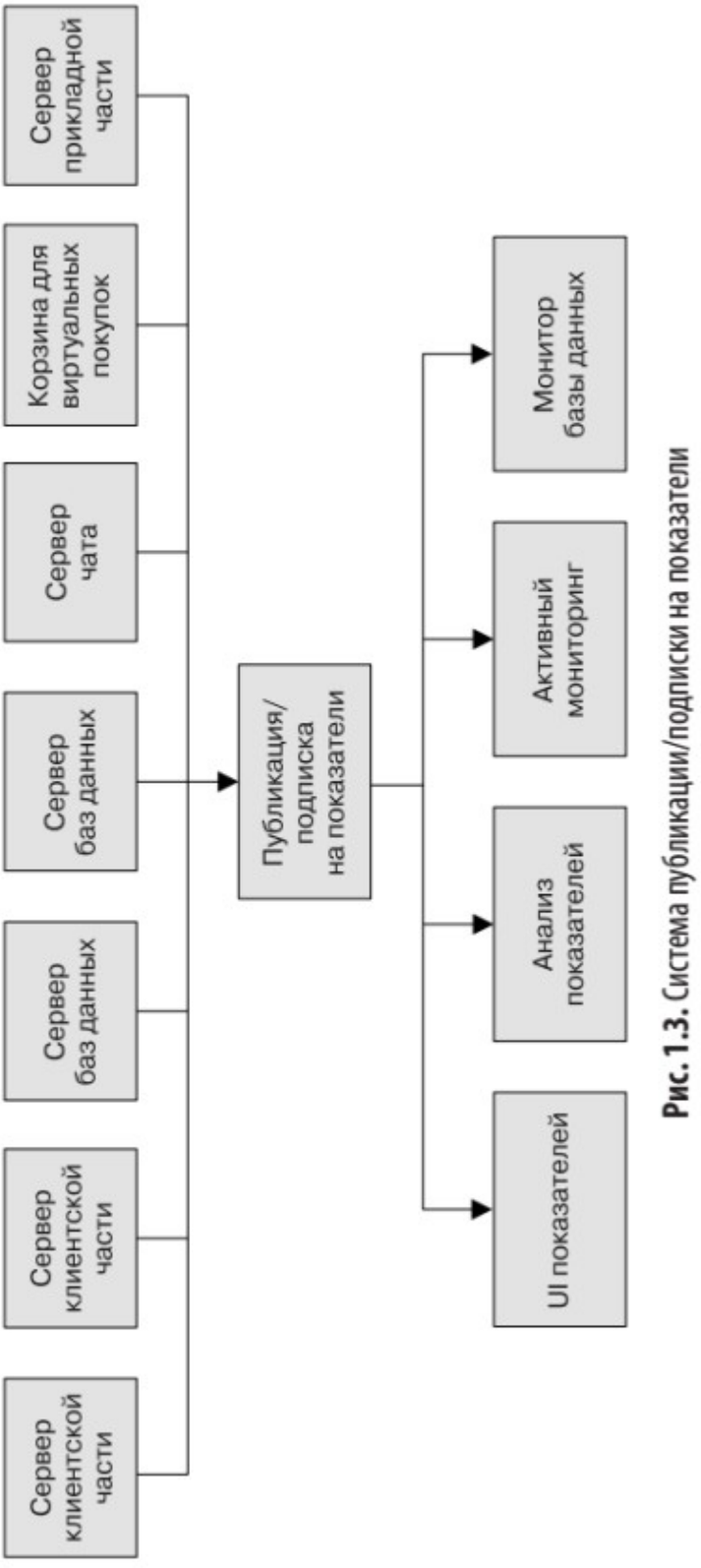


**Рис. 1.3.** Система публикации/подписки на показатели

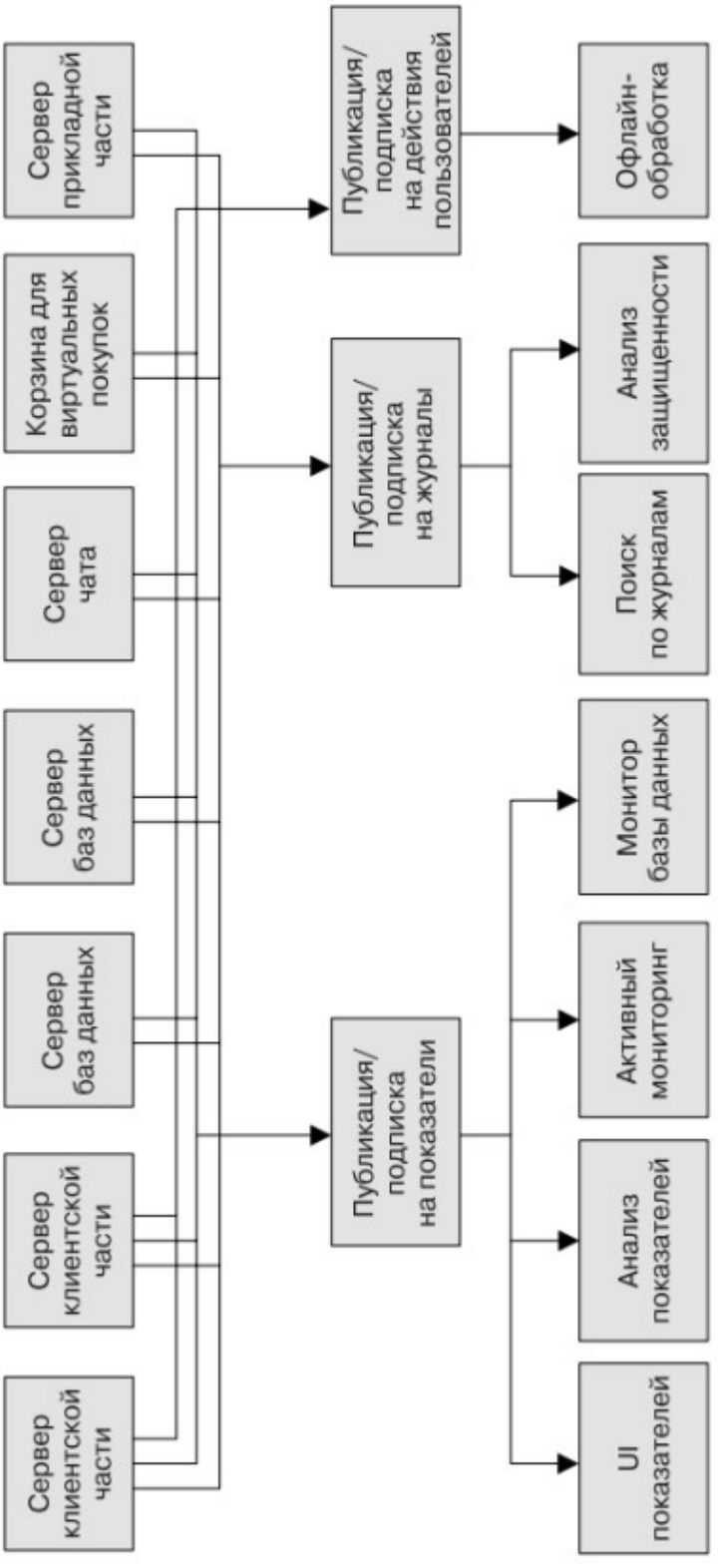


**Рис. 1.4.** Несколько систем публикации/подписки

## Сообщения и пакеты

Используемая в Kafka единица данных называется *сообщением* (message). Если ранее вы работали с базами данных, то можете рассматривать сообщение как аналог *строки* (row) или *записи* (record). С точки зрения Kafka сообщение представляет собой просто массив байтов, так что для нее содержащиеся в нем данные не имеют формата или какого-либо смысла. В сообщении может быть дополнительный фрагмент метаданных, называемый *ключом* (key). Он тоже представляет собой массив байтов и, как и сообщение, не несет для Kafka никакого смысла. Ключи используются при необходимости лучше управлять записью сообщений в разделы. Простейшая схема такова: генерация единообразного хеш-значения ключа с последующим выбором номера раздела для сообщения путем деления этого значения по модулю общего числа разделов в топике. Это гарантирует попадание сообщений с одним ключом в один раздел (при условии, что количество разделов не изменится).

Для большей эффективности сообщения в Kafka записываются пакетами. *Пакет* (batch) представляет собой просто набор сообщений, относящихся к одному топику и разделу. Передача каждого сообщения туда и обратно по сети привела бы к существенному перерасходу ресурсов, а объединение сообщений в пакет эту проблему уменьшает. Конечно, необходимо соблюдать баланс между временем задержки и пропускной способностью: чем больше пакеты, тем больше сообщений можно обрабатывать за единицу времени, но тем дольше распространяется отдельное сообщение. Пакеты обычно сжимаются, что позволяет передавать и хранить данные более эффективно за счет некоторого расхода вычислительных ресурсов. Мы обсудим ключи и пакеты более подробно в главе 3.

## Схемы

Хотя сообщения для Kafka — всего лишь непрозрачные массивы байтов, рекомендуется накладывать на содержимое сообщений дополнительную структуру — схему, которая позволяла бы с легкостью их разбирать. Существует много вариантов задания *схемы* сообщений в зависимости от потребностей конкретного приложения. Упрощенные системы, например нотация объектов JavaScript (JavaScript Object Notation, JSON) и расширяемый язык разметки (Extensible Markup Language, XML), просты в использовании, их удобно читать человеку. Однако им не хватает таких свойств, как надежная работа с типами и совместимость разных версий схемы. Многим разработчикам Kafka нравится Apache Avro — фреймворк сериализации, изначально предназначенный для Hadoop. Avro обеспечивает компактный формат сериализации, схемы, отделенные от содержимого сообщений и не требующие генерации кода при изменении,

а также сильную типизацию данных и эволюцию схемы с прямой и обратной совместимостью.

Для Kafka важен единообразный формат данных, ведь он дает возможность разъединять код записи и чтения сообщений. При тесном сцеплении этих задач приходится модифицировать приложения-подписчики, чтобы они могли работать не только со старым, но и с новым форматом данных. Только после этого можно будет использовать новый формат в публикующих сообщения приложениях. Благодаря применению четко заданных схем и хранению их в общем репозитории сообщения в Kafka можно читать, не координируя действия. Мы рассмотрим схемы и сериализацию подробнее в главе 3.

## Топики и разделы

Сообщения в Kafka распределяются по *топикам* (topics). Ближайшая аналогия — таблица базы данных или каталог файловой системы. Топики, в свою очередь, разбиваются на *разделы* (partitions). Если вернуться к описанию журнала фиксации, то раздел представляет собой отдельный журнал. Сообщения записываются в него путем добавления в конец, а читаются по порядку от начала к концу. Заметим: поскольку топик обычно состоит из нескольких разделов, не гарантируется упорядоченность сообщений в пределах всего топика — лишь в пределах отдельного раздела. На рис. 1.5 показан топик с четырьмя разделами, в конец каждого из которых добавляются сообщения. Благодаря разделам Kafka обеспечивает также избыточность и масштабируемость. Любой из разделов можно разместить на отдельном сервере, что означает возможность горизонтального масштабирования системы на несколько серверов для достижения производительности, далеко выходящей за пределы возможностей одного сервера. Кроме того, разделы могут быть реплицированы, так что на разных серверах будет храниться копия одного и того же раздела на случай выхода из строя одного сервера.


**Рис. 1.5.** Представление топика с несколькими разделами

При обсуждении данных, находящихся в таких системах, как Kafka, часто используется термин *«поток данных»* (stream). Чаще всего он рассматривается отдельным топиком независимо от числа разделов, представляющих собой единый поток данных, перемещающихся от производителей к потребителям. Чаще всего сообщения рассматривают подобным образом при обсуждении потоковой обработки, при которой фреймворки, в частности Kafka Streams, Apache Samza и Storm, работают с сообщениями в режиме реального времени. Принцип их действия подобен принципу работы офлайн-фреймворков, в частности Hadoop, предназначенных для операций с большими данными. Обзор темы потоковой обработки приведен в главе 14.

## Производители и потребители

Пользователи Kafka делятся на два основных типа: производители и потребители. Существуют также продвинутые клиентские API — API Kafka Connect для интеграции данных и Kafka Streams для потоковой обработки. Продвинутые клиенты применяют производители и потребители в качестве строительных блоков, предоставляя на их основе функциональность более высокого уровня.

*Производители* (producers) генерируют новые сообщения. В других системах обмена сообщениями по типу «публикация/подписка» их называют *издателями* (publishers) или *авторами* (writers). Производители сообщений создают их для конкретного топика. По умолчанию производитель будет равномерно поставлять сообщения во все разделы топика. В некоторых случаях он направляет сообщение в конкретный раздел. Для этого обычно служат ключ сообщения и объект `Partitioner`, генерирующий хеш ключа и устанавливающий его соответствие с конкретным разделом. Это гарантирует запись всех сообщений с одинаковым ключом в один и тот же раздел. Производитель может также воспользоваться собственным объектом `Partitioner` со своими бизнес-правилами распределения сообщений по разделам. Более подробно поговорим о производителях в главе 3.

*Потребители* (consumers) читают сообщения. В других системах обмена сообщениями по типу «публикация/подписка» их называют *подписчиками* (subscribers) или *читателями* (readers). Потребитель подписывается на один топик или более и читает сообщения в порядке их создания в каждом разделе. Он отслеживает, какие сообщения он уже прочитал, запоминая смещение сообщений. *Смещение* (offset) — непрерывно возрастающее целочисленное значение — еще один элемент метаданных, который Kafka добавляет в каждое сообщение при его создании. Смещения сообщений в конкретном разделе не повторяются, а следующее сообщение имеет большее смещение (хотя и не обязательно монотонно большее).

Благодаря сохранению следующего возможного смещения для каждого раздела обычно в хранилище самой Kafka потребитель может приостанавливать и возобновлять свою работу, не забывая, в каком месте он читал.

Потребители работают в составе *групп потребителей* (consumer groups) — одного или нескольких потребителей, объединившихся для обработки топика. Организация в группы гарантирует чтение каждого раздела только одним членом группы. На рис. 1.6 представлены три потребителя, объединенные в одну группу для обработки топика. Два потребителя обрабатывают по одному разделу, а третий — два. Соответствие потребителя разделу иногда называют *владением* (ownership) раздела потребителем.

Таким образом, потребители получают возможность горизонтального масштабирования для чтения топиков с большим количеством сообщений. Кроме того, в случае сбоя отдельного потребителя оставшиеся члены группы переназначат потребляемые разделы так, чтобы взять на себя его задачу. Потребители и группы потребителей подробнее описываются в главе 4.


**Рис. 1.6.** Чтение топика группой потребителей

## Брокеры и кластеры

Отдельный сервер Kafka называется *брокером* (broker). Он получает сообщения от производителей, присваивает им смещения и записывает сообщения в дисковое хранилище. Он также обслуживает потребители и отвечает на запросы выборки из разделов, возвращая опубликованные сообщения. В зависимости от конкретного аппаратного обеспечения и его производительности отдельный брокер может с легкостью обрабатывать тысячи разделов и миллионы сообщений в секунду.

Брокеры Kafka предназначены для работы в составе *кластера* (cluster). Один из брокеров кластера функционирует в качестве *контроллера* (cluster controller). Контроллер кластера выбирается автоматически из числа работающих членов кластера. Он отвечает за административные операции, включая распределение разделов по брокерам и мониторинг отказов последних. Каждый раздел принадлежит одному из брокеров кластера, который называется *ведущим* (leader). Реплицированный раздел, как видно на рис. 1.7, можно назначить дополнительным брокерам, которые называются *последователями* (followers) раздела. Репликация обеспечивает избыточность сообщений в разделе, так что в случае сбоя ведущего один из последователей сможет занять его место. Все производители должны соединяться с ведущим для публикации сообщений, но потребители могут получать сообщения либо от ведущего, либо от одного из последователей. Кластерные операции, включая репликацию разделов, подробно рассмотрены в главе 7.

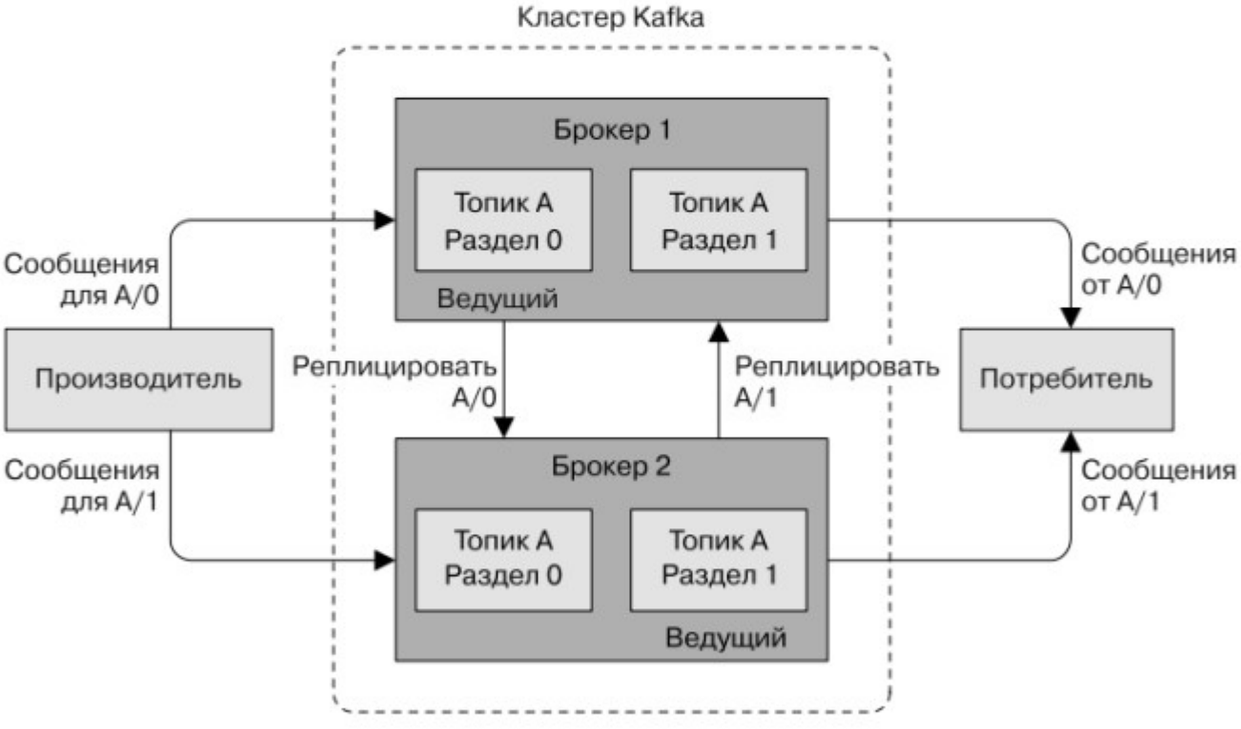


**Рис. 1.7.** Репликация разделов в кластере

Ключевая возможность Apache Kafka — *сохранение информации* (retention) в течение длительного времени. В настройки брокеров Kafka включается длительность хранения топиков по умолчанию — или в течение определенного промежутка времени (например, семь дней), или до достижения разделом определенного размера в байтах (например, 1 Гбайт). Превысившие эти пределы сообщения становятся недействительными и удаляются. Таким образом, на-

стройки сохранения определяют минимальное количество доступной в каждый момент информации. Можно задавать настройки сохранения и для отдельных топиков, чтобы сообщения хранились только до тех пор, пока они нужны. Например, топик для отслеживания действий пользователей можно хранить несколько дней, в то время как параметры приложений — лишь несколько часов. Можно также настроить для топиков вариант хранения *сжатых журналов* (log compacted). При этом Kafka будет хранить лишь последнее сообщение с конкретным ключом. Это может пригодиться для таких данных, как журналы изменений, в случае, когда нас интересует только последнее изменение.

## Несколько кластеров

По мере роста развертываемых систем Kafka может оказаться удобным наличие нескольких кластеров. Вот несколько причин этого.

- Разделение типов данных.
- Изоляция по требованиям безопасности.
- Несколько центров обработки данных (ЦОД) (восстановление в случае катаклизмов).

В ходе работы, в частности, с несколькими ЦОД часто выдвигается требование копирования сообщений между ними. Таким образом онлайн-приложения могут повсеместно получить доступ к информации о действиях пользователей. Например, если пользователь корректирует общедоступную информацию в своем профиле, изменения должны быть видны вне зависимости от ЦОД, в котором отображаются результаты поиска. Или данные мониторинга могут собираться с многих сайтов в одно место, где расположены системы анализа и оповещения. Механизмы репликации в кластерах Kafka предназначены только для работы внутри одного кластера, репликация между несколькими кластерами не осуществляется.

Проект Kafka включает утилиту *MirrorMaker* для репликации данных на другие кластеры. По существу, это просто потребитель и производитель Kafka, связанные воедино очередью. Данная утилита получает сообщения из одного кластера Kafka и публикует их в другом. На рис. 1.8 демонстрируется пример использующей MirrorMaker архитектуры, в которой сообщения из двух локальных кластеров агрегируются в составной кластер, который затем копируется в другие ЦОД. Пускай простота этого приложения не производит у вас ложного впечатления о его возможностях создавать сложные конвейеры данных, которые мы подробнее рассмотрим в главе 9.

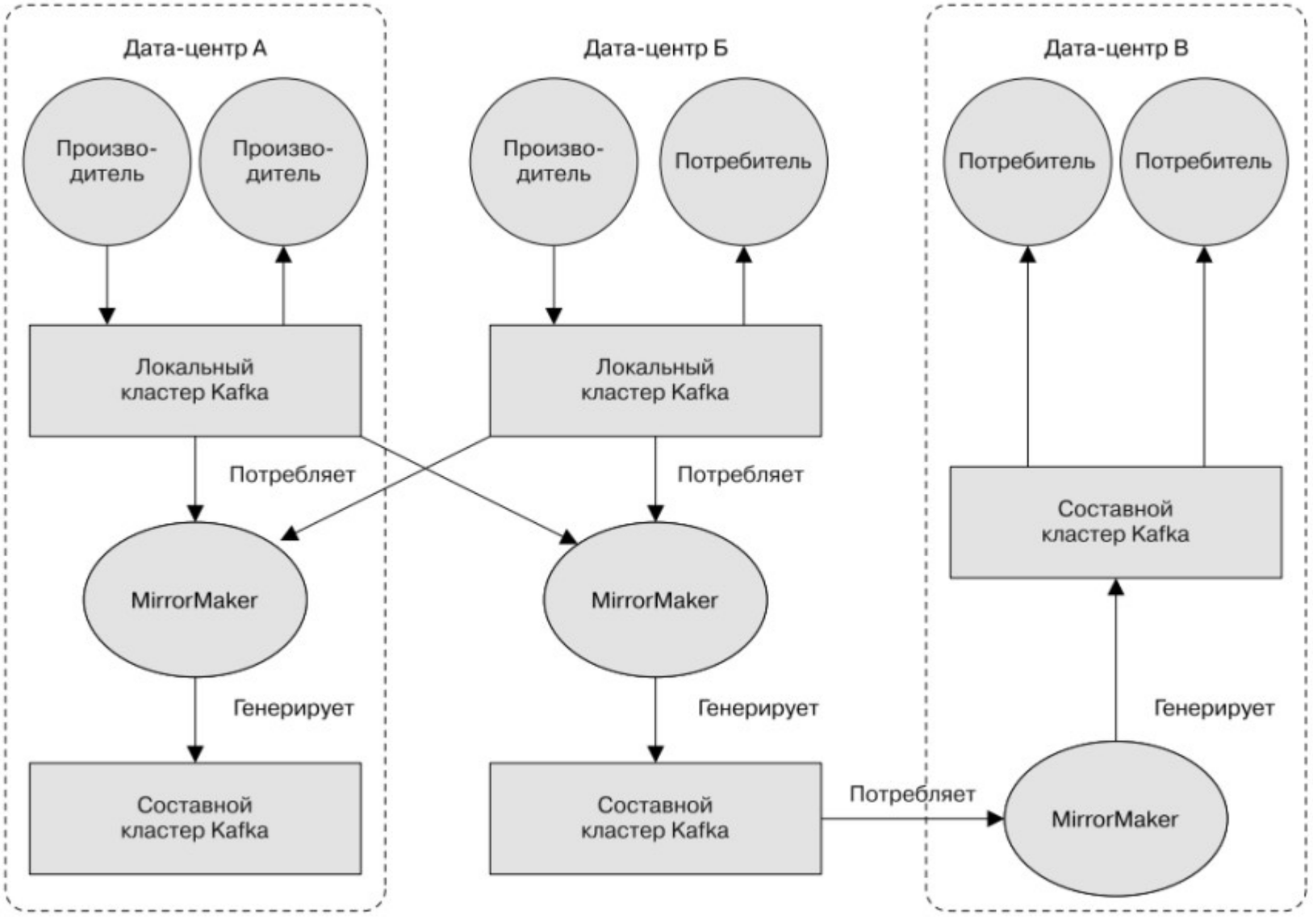


**Рис. 1.8.** Архитектура с несколькими ЦОД