Arenadata™ Streaming

Версия - v1.5-RUS

Руководство разработчика приложений для сервиса Kafka

2020 Arenadata™

Оглавление

1 Kafka Clients **3**
- 1.1 Java . . . 4
- 1.2 C/C++ . . . 5
- 1.3 JMS . . . 5
- 1.4 Python . . . 5
- 1.5 Go . . . 6
- 1.6 .NET . . . 7

2 Kafka Java Consumer **8**
- 2.1 Концепция . . . 8
- 2.2 Конфигурация . . . 9
- 2.3 Инициализация . . . 10
- 2.4 Основные возможности . . . 12
- 2.5 Примеры . . . 15

3 Kafka Java Producer **32**
- 3.1 Концепция . . . 32
- 3.2 Конфигурация . . . 32
- 3.3 Примеры . . . 34

4 Kafka Connect **39**
- 4.1 Connectors & Tasks . . . 39
- 4.2 Partitions & Records . . . 40

В руководстве приведены сведения необходимые разработчику приложений для сервиса Kafka по работе с платформой ADS.

Руководство может быть полезно разработчикам, администраторам, программистам и сотрудникам подразделений информационных технологий, осуществляющих сопровождение платформы.

Important: Контактная информация службы поддержки – e-mail: info@arenadata.io

Глава 1

Kafka Clients

Клиенты для работы с Apache Kafka доступны на многих языках программирования (Clients).

В **ADS** предусмотрена возможность установки клиентских библиотек для различных языков, которые обеспечивают как низкоуровневый доступ к **Kafka**, так и потоковую обработку более высокого уровня.

Таблица1.1.: Клиентские библиотки для различных языков

Ссылка на установку	Ссылка на документацию
C/C++	github.com/edenhill/librdkafka
Go	github.com/confluentinc/confluent-kafka-go
Java	Kafka Java Consumer и Kafka Java Producer
JMS	JMS Client
.NET	github.com/confluentinc/confluent-kafka-dotnet
Python	github.com/confluentinc/confluent-kafka-python

В следующей таблице описана поддержка клиента по различным функциям платформы **ADS**.

Таблица1.2.: Поддержка клиента по различным функциям платформы

Функция	C/C++	Go	Java	.NET	Python
Admin API	Да	Да	Да	Нет	Да
Control Center metrics integration	Да	Да	Да	Да	Да
Custom partitioner	Да	Нет	Да	Нет	Нет
Exactly Once Semantics	Нет	Нет	Да	Нет	Нет
Idempotent Producer	Нет	Нет	Да	Нет	Нет
Kafka Streams	Нет	Нет	Да	Нет	Нет
Record Headers	Да	Да	Да	Да	Да
SASL Kerberos/GSSAPI	Да	Да	Да	Да	Да
SASL PLAIN	Да	Да	Да	Да	Да
SASL SCRAM	Да	Да	Да	Да	Да
Simplified installation	Да	Нет	Да	Да	Да
Schema Registry	Да	Нет	Да	Да	Да
Topic Metadata API	Да	Да	Да	Нет	Да

1.1 Java

Все JAR-файлы, включенные в пакеты, также доступны в репозитории. Далее приведен пример файла *POM*, показывающий, как добавить репозиторий:

```
<repositories>

  <repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
  </repository>

  <!-- further repository entries here -->

</repositories>
```

Репозиторий включает в себя скомпилированные версии **Kafka**. Чтобы сослаться на версию **Kafka 2.0** необходимо использовать в файле *pom.xml* следующее:

```
<dependencies>

  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>2.0.1-cp1</version>
  </dependency>

  <!-- further dependency entries here -->

</dependencies>
```

Important: Arenadata всегда вносит исправления в проект с открытым исходным кодом Apache Kafka. Однако точные версии (и их имена), включаемые в платформу ADS, могут отличаться от артефактов Apache при не совпадении релизов. Если они отличаются, Arenadata сохраняет идентификаторы *groupId* и *artifactId*, но добавляет суффикс *-cpX* (где X – цифра) к идентификатору версии ADS для видимого отличия от артефактов Apache

Есть возможность ссылаться на артефакты для всех библиотек **Java**, которые включены в **ADS**. Например, чтобы использовать сериализаторы с открытым исходным кодом **Arenadata**, которые интегрируются с остальной частью платформы **ADS**, необходимо в *pom.xml* включить следующее:

```
<dependencies>

  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <!-- For Confluent Platform 5.0.1 -->
    <version>5.0.1</version>
  </dependency>

  <!-- further dependency entries here -->

</dependencies>
```

1.2 C/C++

Клиент **C/C++**, называемый *librdkafka*, доступен в виде исходного кода и в виде предварительно скомпилированных двоичных файлов для дистрибутивов **Linux** на основе **Debian** и **Red Hat**, а также **macOS**. Большинство пользователей используют скомпилированные двоичные файлы.

Для дистрибутивов **Linux** необходимо следовать инструкциям для **Debian** или **Red Hat**, а затем использовать *yum* или *apt-get* для установки соответствующих пакетов. Например, разработчику, создающему приложение **C** на дистрибутиве **Red Hat**, рекомендуется использовать пакет *librdkafka-devel*:

```
sudo yum install librdkafka-devel
```

В дистрибутиве **Debian** используется пакет *librdkafka-dev*:

```
sudo apt-get install librdkafka-dev
```

В **macOS** последняя версия доступна через Homebrew:

```
brew install librdkafka
```

Исходный код доступен в архивах *ZIP* и *TAR* в каталоге *src/*.

1.3 JMS

Клиент **JMS** – это библиотека, используемая в приложениях **Java**. Чтобы сослаться на *kafka-jms-client* в проекте для начала необходимо добавить репозиторий в файл *pom.xml*:

```
<repositories>
    <repository>
        <id>confluent</id>
        <url>http://packages.confluent.io/maven/</url>
    </repository>
</repositories>
```

Затем добавить зависимость от клиента **JMS**, а также спецификацию API JMS (при этом заменив *[version]* на требуемую):

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-jms-client</artifactId>
    <version>[version]</version>
</dependency>
<dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jms_1.1_spec</artifactId>
    <version>1.1</version>
</dependency>
```

Можно загрузить JAR-файл JMS-клиента напрямую, перейдя по следующему URL-адресу (при этом заменив *[version]* на требуемую):

```
http://packages.confluent.io/maven/io/confluent/kafka-jms-client/[version]/kafka-jms-client
-[version].jar
```

1.4 Python

Клиент **Python**, именуемый *confluent-kafka-python*, доступен в PyPI. Клиент **Python** использует *librdkafka* клиента **C**. Поэтому для установки **Python** сначала необходимо инсталлировать C, включая его пакет

разработки, а затем установить библиотеку с помощью *pip* (как для **Linux**, так и для **macOS**):

```
pip install confluent-kafka
```

При этом осуществляется глобальная установка пакета для среды **Python**. Для инсталляции клиента только под конкретный проект можно использовать *virtualenv*.

После чего в **Python** можно импортировать библиотеку:

```
from confluent_kafka import Producer

conf = {'bootstrap.servers': 'localhost:9092', 'client.id': 'test', 'default.topic.config': {'acks
↪': 'all'}}
producer = Producer(conf)
producer.produce(topic, key='key', value='value')
```

Исходный код доступен в архивах *ZIP* и *TAR* в каталоге *src/*.

1.5 Go

Клиент **Go**, именуемый *confluent-kafka-go*, распространяется через GitHub и gopkg.in с привязкой к конкретным версиям. Клиент **Go** использует *librdkafka* клиента **C** и представляет его как библиотеку **Go**, используя cgo. Для установки клиента **Go** сначала необходимо инсталлировать клиент **C**, включая его пакет разработки, а также набор инструментов для сборки с *pkg-config*. В дистрибутивах **Linux** на основе **Red Hat** в дополнение к *librdkafka* следует установить следующие пакеты:

```
sudo yum groupinstall "Development Tools"
```

В дистрибутивах на основе **Debian**, помимо *librdkafka*, необходимо установить:

```
sudo apt-get install build-essential pkg-config git
```

В **macOS** с помощью Homebrew установить:

```
brew install pkg-config git
```

Далее использовать *go get* для установки библиотеки:

```
go get gopkg.in/confluentinc/confluent-kafka-go.v0/kafka
```

Код **Go** теперь может импортировать и использовать клиент. Также можно собрать и запустить небольшую утилиту командной строки **go-kafkacat**, чтобы убедиться, что установка прошла успешно:

```
go get gopkg.in/confluentinc/confluent-kafka-go.v0/examples/go-kafkacat
$GOPATH/bin/go-kafkacat --help
```

Для настройки статической ссылки к *librdkafka* необходимо добавть флаг *-tags static* к командам *go get*. Это позволяет статически связать саму *librdkafka*, чтобы ее динамическая библиотека не требовалась в целевой системе развертывания. Однако при этом статически связанные зависимости *librdkafka* (такие как *ssl*, *sasl2*, *lz4* и пр.), остаются по-прежнему динамически связанными и требуются в целевой системе. Экспериментальная опция для создания полностью статически связанного двоичного файла также доступна – использование флага *-tags static_all*. При этом требуется, чтобы все зависимости были доступны как статические библиотеки (например, *libsasl2.a*). Статические библиотеки обычно не устанавливаются по умолчанию, но доступны в соответствующих пакетах *-dev* или *-devel* (например, *libsasl2-dev*).

Исходный код доступен в архивах *ZIP* и *TAR* в каталоге *src/*.

1.6 .NET

Клиент **.NET**, именуемый *confluent-kafka-dotnet*, доступен в NuGet. Клиент **.NET** использует *librdkafka* клиента **C**. Предварительно скомпилированные двоичные файлы для *librdkafka* предоставляются через зависимый пакет **NuGet** librdkafka.redist для ряда популярных платформ (win-x64, win-x86, debian-x64, rhel-x64 и osx).

Для того, чтобы сослаться на *confluent-kafka-dotnet* из проекта, необходимо выполнить следующую команду в консоли диспетчера пакетов:

```
PM> Install-Package Confluent.Kafka
```

Important: Зависимый пакет *librdkafka.redist* устанавливается автоматически

Для того, чтобы сослаться на *confluent-kafka-dotnet* в файле *project.json*, необходимо включить следующую ссылку в раздел зависимостей:

```
"dependencies": {
    ...
    "Confluent.Kafka": "0.9.4"
    ...
}
```

И затем выполнить команду `dotnet restore`, чтобы восстановить зависимости проекта через **NuGet**.

Клиент *confluent-kafka-dotnet* предназначен для платформ **net451** и **netstandard1.3** и поддерживается в **.NET Framework** версии *4.5.1* и выше и **.NET Core** версии *1.0* (в **Windows**, **Linux** и **Mac**) и выше. Не поддерживается на **Mono**.

Глава 2

Kafka Java Consumer

Платформа **ADS** включает в себя Java consumer, поставляемый вместе с **Kafka**.

В документе представлен общий обзор работы потребителя, введение в параметры конфигурации для настройки и примеры из каждой клиентской библиотеки.

2.1 Концепция

Consumer group – группа потребителей, взаимодействующих для использования данных из топиков. Партиции в топиках делятся между потребителями в группе, и при изменениях в группе партиции перераспределяются таким образом, что каждый потребитель получает пропорциональную долю партиций. Такой процесс называется перебалансировкой группы (rebalancing the group).

Основное различие в управлении группами между старым “high-level” потребителем и новым заключается в том, что первый зависит от **ZooKeeper**, а второй использует групповой протокол, встроенный в саму **Kafka**. В данном протоколе один из брокеров назначается координатором группы и отвечает за управление ее потребителями и за назначение им партиций.

Координатор каждой группы выбирается из лидеров внутренних смещений $__consumer_offsets$. Обычно идентификатор группы хэшируется в одной из партиций топика, и лидер данной партиции выбирается в качестве координатора. Таким образом, управление группами потребителей разделяется примерно поровну между всеми брокерами в кластере, что позволяет масштабировать количество групп за счет увеличения числом брокеров.

Когда потребитель запускается, он находит координатора для своей группы и отправляет запрос на присоединение. При этом координатор начинает перебалансировку, что приводит к формированию новой группы.

Каждый участник в группе должен посылать heartbeat-сообщения координатору. В случае если до истечения настроенного тайм-аута сессии такового не получено, координатор исключает потребителя из группы и переназначает его партиции.

2.1.1 Управление смещением (Offset Management)

После получения потребителем назначения от координатора необходимо выявить начальную позицию для каждой определенной партиции. Когда группа создается впервые, до того, как какие-либо сообщения были использованы, позиция устанавливается в соответствии с политикой сброса смещения ($auto.offset.reset$). Как правило, потребление начинается с самого раннего либо с самого позднего смещения.

Потребителю необходимо фиксировать свои смещения в партиции в соответствии с ходом прочтения сообщений. Поскольку, если потребитель выходит из строя или выключается, его партиции переназначаются другому участнику группы, который начинает потребление сообщений с последнего закоммиченного смещения.

В случае аварийного завершения работы потребитиеля до того, как какое-либо его смещение зафиксировано, следующий потребитель использует политику сброса.

Политика фиксации смещения имеет ключевое значение для обеспечения необходимых приложению гарантий доставки сообщений. По умолчанию потребитель настроен на использование политики автоматического коммита, которая инициирует фиксацию с периодическим интервалом. Также потребителем поддерживается API, который можно использовать для ручного управления смещением. В примерах приведено несколько подробных случаев API-фиксации и обсуждение компромиссов с точки зрения производительности и надежности (*Примеры*).

При записи во внешнюю систему позиция потребителя должна быть согласована с тем, что хранится в виде выходных данных. Именно поэтому потребитель хранит свое смещение в том же месте, где выходные данные. Например, Kafka Connect записывает данные в **HDFS** вместе со смещениями считываемых данных, что гарантирует обоюдное обновление данных и смещений. Аналогичная схема применяется для многих других систем данных, требующих более строгой семантики, и для которых сообщения не имеют первичного ключа для обеспечения дедупликации.

Так **Kafka** поддерживает обработку exactly-once в Kafka Streams, и поставщик или потребитель транзакций может использоваться для обеспечения доставки exactly-once при передаче и обработке данных между топиками **Kafka**. В противном случае **Kafka** гарантирует доставку at-least-once по умолчанию, но при этом можно реализовать доставку at-most-once, отключив повторные попытки для поставщика и зафиксировав смещения в потребителе перед обработкой пакета сообщений.

2.2 Конфигурация

Полный список параметров конфигурации доступен в документе Настройки платформы Arenadata Streaming. Но некоторые из ключевых параметров и их влияние на поведение потребителя описаны в текущей главе.

2.2.1 Базовая конфигурация (Core Configuration)

Единственная обязательная настройка – это *bootstrap.servers*, но при этом должен быть установлен *client.id* для сопоставления запросов в брокере со сделавшим их инстансом клиента. Как правило, все потребители в одной группе используют один и тот же идентификатор клиента.

2.2.2 Конфигурация группы (Group Configuration)

Параметр *group.id* должен быть всегда настроен за исключением случаев, когда API используется просто по назначению и нет необходимости хранить смещения в **Kafka**.

Тайм-аутом сессии можно управлять в параметре *session.timeout.ms*. Значение по умолчанию установлено на *30 секунд*, но если приложению требуется больше времени для обработки сообщений, то для избежания чрезмерной перебалансировки значение параметра можно безопасно увеличить. Это в основном актуально при использовании Java consumer и обработке сообщений в одном потоке. В таком случае также можно регулировать параметр *max.poll.records* для настройки количества требуемых для обработки на каждой итерации цикла записей (более подробно вопрос рассмотрен в главе *Основные возможности*).

Основным недостатком применения долгого тайм-аута сессии является то, что координатору требуется больше времени для обнаружения сбоя инстанса потребителя, а это значит, что другому потребителю в группе требуется больше времени для передачи партиций. Но при этом в случае необходимости нормального выключения потребитель отправляет координатору явный запрос покинуть группу, который инициирует немедленную перебалансировку.

Другим параметром, влияющим на поведение перебалансировки, является *heartbeat.interval.ms*. Он контролирует, как часто потребитель должен отправлять heartbeats-сообщения координатору. Это также способ, когда необходимость перебалансировки определяется засчет потребителя, поэтому более короткий интервал

heartbeats-сообщений обычно означает более быструю перебалансировку. Значение по умолчанию составляет *3 секунды*. Для больших групп целесообразно увеличить значение параметра.

2.2.3 Управление смещением (Offset Management)

Двумя основными параметрами, влияющими на управление смещением, являются автоматическая фиксация и политика сброса смещения. В первом случае при установленном по умолчанию параметре *enable.auto.commit* потребитель автоматически фиксирует смещения с заданным в *auto.commit.interval.ms* интервалом (по умолчанию – *5 секунд*).

Второй параметр *auto.offset.reset* определяет поведение потребителя, когда нет зафиксированной позиции смещения (в случае, когда группа инициализируется впервые), или когда оно выходит за пределы диапазона. Можно установить сброс положения на самое раннее смещение – *earliest*, либо на самое позднее – *latest* (задано по умолчанию). Также можно выбрать значение *none* для самостоятельной установки начального смещения и ручной обработки ошибок вне диапазона.

2.3 Инициализация

Потребитель Java consumer создается с помощью стандартного файла свойств *Properties*:

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("group.id", "foo");
config.put("bootstrap.servers", "host1:9092,host2:9092");
new KafkaConsumer<K, V>(config);
```

Important: Ошибки конфигурации приводят к возникновению исключения *KafkaException* из конструктора *KafkaConsumer*

Конфигурация **C/C++** (*librdkafka*) похожа, но при этом необходимо обрабатывать ошибки конфигурации непосредственно при настройке свойств:

```
char hostname[128];
char errstr[512];

rd_kafka_conf_t *conf = rd_kafka_conf_new();

if (gethostname(hostname, sizeof(hostname))) {
 fprintf(stderr, "%% Failed to lookup hostname\n");
 exit(1);
}

if (rd_kafka_conf_set(conf, "client.id", hostname,
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
 fprintf(stderr, "%% %s\n", errstr);
 exit(1);
}

if (rd_kafka_conf_set(conf, "group.id", "foo",
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
 fprintf(stderr, "%% %s\n", errstr);
 exit(1);
}

if (rd_kafka_conf_set(conf, "bootstrap.servers", "host1:9092,host2:9092",
```

```
                    errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
 fprintf(stderr, "%% %s\n", errstr);
 exit(1);
}

/* Create Kafka consumer handle */
rd_kafka_t *rk;
if (!(rk = rd_kafka_new(RD_KAFKA_CONSUMER, conf,
                        errstr, sizeof(errstr)))) {
 fprintf(stderr, "%% Failed to create new consumer: %s\n", errstr);
 exit(1);
}
```

Клиент **Python** может быть настроен через словарь следующим образом:

```
from confluent_kafka import Consumer

conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'group.id': "foo",
        'default.topic.config': {'auto.offset.reset': 'smallest'}}

consumer = Consumer(conf)
```

Клиент **Go** использует объект *ConfigMap* для передачи конфигурации потребителю:

```
import (
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

consumer, err := kafka.NewConsumer(&kafka.ConfigMap{
     "bootstrap.servers":    "host1:9092,host2:9092",
     "group.id":             "foo",
     "default.topic.config": kafka.ConfigMap{"auto.offset.reset": "smallest"}})
```

В **C#** используется *Dictionary<string, object>*:

```
using System.Collections.Generic;
using Confluent.Kafka;

...

var config = new Dictionary<string, object>
{
    { "bootstrap.servers", "host1:9092,host2:9092" },
    { "group.id", "foo" },
    { "default.topic.config", new Dictionary<string, object>
        {
            { "auto.offset.reset", "smallest" }
        }
    }
}

using (var consumer = new Consumer<Null, string>(config, null, new StringDeserializer(Encoding.
↪UTF8)))
{
    ...
}
```

2.4 Основные возможности

Хотя Java-клиент и *librdkafka* имеют много общих опций конфигурации и базовых функций, они используют довольно разные подходы, когда дело доходит до их потоковой модели и работы с потребителями. Прежде чем углубляться в примеры, полезно разобраться в дизайне API каждого клиента.

2.4.1 Java Client

Java Client разработан вокруг цикла обработки событий под управляением *poll()* API. Конструкция мотивирована системными вызовами UNIX *select* и *poll*. Базовый цикл потребления с Java API обычно принимает следующую форму:

```
while (running) {
  ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
  process(records); // application-specific processing
  consumer.commitSync();
}
```

В Java consumer нет фонового потока. API зависит от вызовов *poll()* для управления всеми операциями ввода-вывода, включая:

- Присоединение к группе потребителей и обработка перебалансировкой партиций;
- Периодичная отправка heartbeats-сообщений;
- Периодичная отправка зафиксированных смещений (при включенном автокоммите);
- Отправка и получение запросов на выборку для назначенных партиций.

Такая однопоточная модель означает, что нельзя отправлять heartbeats-сообщения, пока приложение обрабатывает записи по вызову *poll()*. Это приводит к тому, что потребитель выпадает из группы, если цикл обработки событий завершается, либо если задержка в обработке записи приводит к истечению времени ожидания сессии до следующей итерации цикла. Так и было задумано. Одна из проблем, которую пытается решить **Java Client**, – обеспечение жизнедеятельности потребителей в группе. В то время, пока потребителю назначены партиции, другие члены группы не могут их же использовать, поэтому важно убедиться, что каждый конкретный потребитель действительно прогрессирует.

Данная функция защищает приложение от большого класса сбоев, но недостатком является то, что необходима настройка времени ожидания сессии так, чтобы потребитель не превышал его в своей обычной обработке записей. Кроме этого параметр *max.poll.records* устанавливает верхнюю границу количества записей, возвращаемых при каждом вызове. Поэтому важно использовать *poll()* и *max.poll.records* с достаточно большим тайм-аутом сессии (например, от *30* до *60 секунд*) и ограничивать количество обработанных записей на каждой итерации.

В случае если данные параметры не настроены надлежащим образом, это, как правило, приводит к исключению *CommitFailedException* смещения для обработанных записей. При использовании политики автоматической фиксации, можно даже не заметить, когда это происходит, так как потребитель молча игнорирует сбои коммитов (если только это не происходит достаточно часто, чтобы повлиять на показатели задержки). Исключение можно перехватить и либо проигнорировать, либо выполнить необходимую логику отката:

```
while (running) {
  ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
  process(records); // application-specific processing
  try {
    consumer.commitSync();
  } catch (CommitFailedException e) {
    // application-specific rollback of processed records
```

```
  }
}
```

2.4.2 C/C++ Client (librdkafka)

Librdkafka использует многопоточный подход к потреблению **Kafka**. С точки зрения пользователя, взаимодействие с API не слишком отличается от примера, используемого Java-клиентом, когда пользователь вызывает *rd_kafka_consumer_poll* в цикле, хотя данный API возвращает только одно сообщение или событие за раз:

```
while (running) {
 rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
 if (rkmessage) {
   msg_process(rkmessage);
   rd_kafka_message_destroy(rkmessage);

   if ((++msg_count % MIN_COMMIT_COUNT) == 0)
     rd_kafka_commit(rk, NULL, 0);
 }
}
```

В отличие от Java-клиента, *librdkafka* выполняет всю выборку и координирует взаимодействие в фоновых потоках, что освобождает от сложности настройки тайм-аута сессии в соответствии с ожидаемым временем обработки. Однако, поскольку фоновый поток поддерживает потребителя до тех пор, пока клиент не закроется, важно убедиться, что процесс не простаивает, так как в этом случае он продолжает удерживать назначенные ему партиции.

При этом перебалансировка партиций также происходит в фоновом потоке, а это говорит о том, что все равно приходится обрабатывать потенциальные ошибки коммита, поскольку потребитель может больше не иметь того же назначения партиций, когда начинается фиксация. Это не требуется при включенном автокоммите, так как при этом ошибки коммита игнорируются в автоматическом режиме, но это также означает, что нет возможности откатить обработку.

```
while (running) {
  rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 1000);
  if (!rkmessage)
    continue; // timeout: no message

  msg_process(rkmessage); // application-specific processing
  rd_kafka_message_destroy(rkmessage);

  if ((++msg_count % MIN_COMMIT_COUNT) == 0) {
    rd_kafka_resp_err_t err = rd_kafka_commit(rk, NULL, 0);
    if (err) {
      // application-specific rollback of processed records
    }
  }
}
```

2.4.3 Python, Go и .NET Clients

Клиенты **Python**, **Go** и **.NET** на внутреннем уровне используют *librdkafka*, поэтому у них также применяется многопоточный подход к потреблению **Kafka**. С точки зрения пользователя, взаимодействие с API не слишком отличается от примера, используемого Java-клиентом, когда пользователь вызывает метод *poll()* в цикле, хотя данный API возвращает только одно сообщение за раз.

Python:

```
try:
    msg_count = 0
    while running:
        msg = consumer.poll(timeout=1.0)
        if msg is None: continue

        msg_process(msg) # application-specific processing
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0:
            consumer.commit(async=False)
finally:
    # Shut down consumer
    consumer.close()
```

Go:

```
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        // application-specific processing
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

Поведение потребителя **C#** аналогично, за исключением того, что перед входом в цикл *Poll* необходимо настроить обработчики для различных типов событий, что эффективно делается внутри метода *Poll* (важно обратить внимание, что весь код выполняется в том же потоке):

```
consumer.OnMessage += (_, msg) =>
{
    // handle message.
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}

while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

2.5 Примеры

Далее приведены подробные примеры использования consumer API с особым вниманием к управлению смещением и семантике доставки.

2.5.1 Basic Poll Loop

API потребителя сосредоточен вокруг метода *poll()* для получения записей от брокеров и метода *subscribe()* для выбора топиков. Как правило, потребитель первоначально обращается к методу *subscribe()* для настройки интересующих топиков, а затем запускает цикл *poll()* до завершения работы приложения.

Потребитель намеренно избегает конкретной модели потоков, так как это не безопасно для многопоточного доступа и не дает возможности наличия собственных фоновых потоков. В частности, это означает, что все операции ввода-вывода происходят в потоке, вызыванном методом *poll()*. В приведенном ниже примере цикл опроса заключен в *Runnable*, который упрощает использование с *ExecutorService*:

```
public abstract class BasicConsumeLoop implements Runnable {
  private final KafkaConsumer<K, V> consumer;
  private final List<String> topics;
  private final AtomicBoolean shutdown;
  private final CountDownLatch shutdownLatch;

  public BasicConsumeLoop(Properties config, List<String> topics) {
    this.consumer = new KafkaConsumer<>(config);
    this.topics = topics;
    this.shutdown = new AtomicBoolean(false);
    this.shutdownLatch = new CountDownLatch(1);
  }

  public abstract void process(ConsumerRecord<K, V> record);

  public void run() {
    try {
      consumer.subscribe(topics);

      while (!shutdown.get()) {
        ConsumerRecords<K, V> records = consumer.poll(500);
        records.forEach(record -> process(record));
      }
    } finally {
      consumer.close();
      shutdownLatch.countDown();
    }
  }

  public void shutdown() throws InterruptedException {
    shutdown.set(true);
    shutdownLatch.await();
  }
}
```

В примере жестко запрограммировано время ожидания опроса на *500 миллисекунд*, то есть, если никаких записей не получено до истечения тайм-аута, *poll()* возвращает пустой набор записей. В случае если обработка сообщений связана с дополнительными затратами на настройку, можно добавить проверку ярлыков.

Для отключения потребителя добавляется флаг, который проверяется на каждой итерации цикла. При этом потребитель ожидает *500 миллисекунд* (плюс время обработки сообщения) перед завершением работы. Лучший подход представлен далее в примере.

Важно обратить внимание, что всегда следует вызывать *close()* после завершения работы потребителя. Это обеспечивает закрытие активных сокетов и очистку внутреннего состояния. Также это немедленно инициирует перебалансировку группы, что в свою очередь гарантирует переназначение всех принадлежащих данному потребителю партиций другому члену группы. Если не выполнить закрытие должным образом, брокер инициирует перебалансировку только после истечения времени ожидания сессии. В примере добавлена защелка (latch) для того, чтобы у потребителя было время завершить закрытие перед выключением.

Этот же пример выглядит аналогично в *librdkafka*:

```
static int shutdown = 0;
static void msg_process(rd_kafka_message_t message);

void basic_consume_loop(rd_kafka_t *rk,
                        rd_kafka_topic_partition_list_t *topics) {
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }

  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage) {
      msg_process(rkmessage);
      rd_kafka_message_destroy(rkmessage);
    }
  }

  err = rd_kafka_consumer_close(rk);
  if (err)
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

В **Python**:

```
running = True

def basic_consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
    finally:
        # Close down consumer to commit final offsets.
```

```
        consumer.close()

def shutdown():
    running = False
```

В **Go**:

```
err = consumer.SubscribeTopics(topics, nil)

for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))
    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}

consumer.Close()
```

В **C#**:

```
using (var consumer = new Consumer<Null, string>(config, null, new StringDeserializer(Encoding.
↪UTF8)))
{
    consumer.OnMessage += (_, msg)
        => Console.WriteLine($"Message value: {msg.Value}");

    consumer.OnPartitionEOF += (_, end)
        => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

    consumer.OnError += (_, error)
    {
        Console.WriteLine($"Error: {error}");
        cancelled = true;
    }

    consumer.Subscribe(topics);

    while (!cancelled)
    {
        consumer.Poll(TimeSpan.FromSeconds(1));
    }
}
```

Хотя API-интерфейсы схожи, клиенты **C**/**C++**, **Python**, **Go** и **C#** используют другой подход, нежели **Java**. В то время как потребитель **Java** выполняет все операции ввода-вывода и обработку в потоке переднего плана (foreground thread), остальные клиенты используют фоновый поток (background thread). Основным следствием использования многопоточности (multiple threads) является то, что вызов *rd_kafka_consumer_poll* или *Consumer.poll()* абсолютно безопасен, то есть можно распараллеливать обработку сообщений по нескольким потокам. С высокого уровня опрос извлекает сообщения из очереди, которая заполняется в фоновом потоке.

Другим следствием использования фонового потока является то, что в нем выполняются все heartbeats-сообщения и перебалансировки. Преимущество заключается в отсутствии беспокойства об обработке сообщений, которая может стать следствием “пропуска” потребителем перебалансировки. Недостатком является то, что фоновый поток продолжает отправку heartbeats-сообщений, даже если процессор сообщений умер. А в таком случае потребитель удерживает свои партиции, и задержка чтения продолжается до выключения процесса.

Хотя клиенты используют различные подходы, они не так далеки друг от друга, как кажется. Для обеспечения той же абстракции в клиенте **Java** можно поместить очередь между циклом опроса и процессором сообщений, тогда poll loop заполняет очередь, а процессоры по ней извлекают сообщения.

2.5.2 Shutdown и Wakeup

Альтернативным шаблоном для цикла опроса в Java-клиенте является использование *Long.MAX_ VALUE* для тайм-аута. Для выхода из цикла можно использовать метод потребителя *wakeup()* из отдельного потока. Это вызывает исключение *WakeupException* из потока, блокирующего *poll()*. Если поток блокируется некорректно, то это приводит к вызову следующего опроса.

```
public abstract class ConsumeLoop implements Runnable {
  private final KafkaConsumer<K, V> consumer;
  private final List<String> topics;
  private final CountDownLatch shutdownLatch;

  public BasicConsumeLoop(KafkaConsumer<K, V> consumer, List<String> topics) {
    this.consumer = consumer;
    this.topics = topics;
    this.shutdownLatch = new CountDownLatch(1);
  }

  public abstract void process(ConsumerRecord<K, V> record);

  public void run() {
    try {
      consumer.subscribe(topics);

      while (true) {
        ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
        records.forEach(record -> process(record));
      }
    } catch (WakeupException e) {
      // ignore, we're closing
    } catch (Exception e) {
      log.error("Unexpected error", e);
    } finally {
      consumer.close();
      shutdownLatch.countDown();
    }
  }

  public void shutdown() throws InterruptedException {
    consumer.wakeup();
    shutdownLatch.await();
  }
}
```

2.5.3 Синхронные коммиты

В предыдущих примерах предполагается, что потребитель настроен на автоматическую фиксацию смещений (по умолчанию). Auto-commit в основном работает как cron с периодом, установленным через свойство конфигурации *auto.commit.interval.ms*. Если потребитель аварийно завершает работу, то после перезапуска или перебалансировки положение всех принадлежащих ему партиций сбрасывается до последнего зафиксированного смещения. При этом последний коммит может быть таким же старым, как и сам интервал автоматической фиксации. Любые сообщения, поступившие с момента последнего коммита, необходимо прочитать повторно.

Очевидно, что для сокращения окна дубликатов можно уменьшить интервал автоматической фиксации, но некоторым пользователям может потребоваться еще более точный контроль над смещениями. Поэтому потребитель поддерживает commit API, который дает полный контроль над смещениями. Самый простой и надежный способ ручной фиксации смещений – использовать синхронную фиксацию с помощью *commitSync()*, вызов которой блокирует поток до успешно выполненного коммита.

При непосредственном использовании API фиксации необходимо сначала отключить автоматический коммит в конфигурации, установив для свойства *enable.auto.commit* значение *false*.

```
private void doCommitSync() {
  try {
    consumer.commitSync();
  } catch (WakeupException e) {
    // we're shutting down, but finish the commit first and then
    // rethrow the exception so that the main loop can exit
    doCommitSync();
    throw e;
  } catch (CommitFailedException e) {
    // the commit failed with an unrecoverable error. if there is any
    // internal state which depended on the commit, you can clean it
    // up here. otherwise it's reasonable to ignore the error and go on
    log.debug("Commit failed", e);
  }
}

public void run() {
  try {
    consumer.subscribe(topics);

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      doCommitSync();
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
    shutdownLatch.countDown();
  }
}
```

В данном примере блок *try/catch* добавлен к вызову *commitSync*. Когда фиксация не может быть завершена по причине перебалансировки группы, выдается *CommitFailedException*. Это главное, что с осторожностью необходимо соблюдать при использовании клиента **Java**. Поскольку все сетевые операции ввода-вывода (включая heartbeats-сообщения) и обработка сообщений выполняются в потоке переднего плана, тайм-аут сессии может истечь во время обработки пакета сообщений. Чтобы справиться с этим, есть два

варианта.

В первом варианте сначала можно настроить параметр *session.timeout.ms*, чтобы у обработчика было достаточно времени для завершения обработки сообщений. Затем можно настроить *max.partition.fetch.bytes*, чтобы ограничить объем данных, возвращаемых в одном пакете, но при этом приходится учитывать, сколько партиций содержится в подписанных топиках.

Второй вариант заключается в обработке сообщений в отдельном потоке, но тогда приходится управлять потоком передачи данных, чтобы потоки не отставали. Например, простого помещения сообщений в очередь блокировки, вероятно, недостаточно, если скорость обработки не поспевает за скоростью доставки (в этом случае может не понадобиться отдельный поток). Это может даже усугубить проблему, если цикл опроса заблокирован при вызове метода *offer()*, так как тогда фоновый поток обрабатывает еще больший пакет сообщений. API **Java** предлагает метод *pause()*, чтобы помочь в подобных ситуациях.

В данном случае необходимо установить *session.timeout.ms* достаточно большим, чтобы сбои при перебалансировках происходили реже. Как упомянуто выше, единственным недостатком этого является более длительная задержка переназначения партиций в случае серьезного сбоя (когда потребитель не может быть чисто завершен с помощью *close()*), что на практике редко происходит.

Важно проявить осторожность, так как функция *wakeup()* может быть запущена, пока коммит находится в состоянии ожидания. Рекурсивный вызов безопасней, поскольку инициирует wakeup только один раз.

В **C/C++** (*librdkafka*) можно получить похожее поведение с *rd_kafka_commit*, который используется как для синхронных, так и для асинхронных фиксаций. Однако подход немного отличается, поскольку *rd_kafka_consumer_poll* возвращает отдельные сообщения вместо пакетов, как это делает потребитель **Java**.

```
void consume_loop(rd_kafka_t *rk,
                  rd_kafka_topic_partition_list_t *topics) {
  static const int MIN_COMMIT_COUNT = 1000;

  int msg_count = 0;
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }

  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage) {
      msg_process(rkmessage);
      rd_kafka_message_destroy(rkmessage);

      if ((++msg_count % MIN_COMMIT_COUNT) == 0)
        rd_kafka_commit(rk, NULL, 0);
    }
 }

  err = rd_kafka_consumer_close(rk);
  if (err)
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

В данном примере синхронная фиксация запускается каждые *1000* сообщений. Вторым аргументом *rd_kafka_commit* является список смещений, которые должны быть зафиксированы; при значении *NULL* *librdkafka* фиксирует последние смещения для назначенных позиций. Третий аргумент в *rd_kafka_commit* –

флаг, который определяет асинхронность вызова. Коммит также можно активировать по истечению тайм-аута, чтобы убедиться, что зафиксированная позиция регулярно обновляется.

Поскольку клиент **Python** внутренне использует *librdkafka*, он применяет аналогичный шаблон, устанавливая параметр *async* для вызова метода *Consumer.commit()*. Этот метод также может принимать взаимоисключающие смещения параметров ключевых слов для явного перечисления смещений каждой назначенной партиции топика и *message*, которые фиксируют смещения относительно объекта *Message*, возвращаемого функцией *poll()*.

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(async=False)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

Клиент **Go** также внутренне использует *librdkafka*, поэтому он применяет похожий шаблон, но обеспечивает при этом только синхронный вызов метода *Commit()*. Другие варианты методов фиксации также принимают список смещений для коммитов или *Message*, чтобы зафиксировать смещения относительно считываемого сообщения. При использовании ручного коммита важно отключить конфигурацию *enable.auto.commit*.

```
msg_count := 0
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0 {
            consumer.Commit()
        }
        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
```

```
        fmt.Printf("Ignored %v\n", e)
    }
}
```

Клиент **C#** обеспечивает метод *CommitAsync* с возможными перегрузками. Его можно использовать синхронно, отвечая *Result* или *Wait()* на возвращаемый *Task*. Существуют варианты, которые фиксируют все смещения в текущем назначении, конкретный список смещений или смещение на основе *Message*.

```
var msgCount = 0;

consumer.OnMessage += (_, msg) =>
{
    msgCount += 1;
    if (msgCount % MIN_COMMIT_COUNT == 0)
    {
        consumer.CommitAsync().Wait();
    }
    Console.WriteLine($"Message value: {msg.Value}");
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}

while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

Использование автоматической фиксации обеспечивает доставку “at least once”: **Kafka** гарантирует, что ни одно сообщение не будет пропущено, но возможны дубликаты. В предыдущем примере обеспечивается такая доставка, поскольку фиксация следует за обработкой сообщения. Однако, изменив запрос, можно получить доставку “at most once”. Но при этом следует быть осторожнее с ошибкой коммита, для этого необходимо изменить *doCommitSync*, чтобы он возвращал информацию об успешности транзакции. Так же при синхронной фиксации отменяется необходимость в перехвате исключения *WakeupException*.

```
private boolean doCommitSync() {
  try {
    consumer.commitSync();
    return true;
  } catch (CommitFailedException e) {
    // the commit failed with an unrecoverable error. if there is any
    // internal state which depended on the commit, you can clean it
    // up here. otherwise it's reasonable to ignore the error and go on
    log.debug("Commit failed", e);
    return false;
  }
}

public void run() {
  try {
    consumer.subscribe(topics);
```

```
    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      if (doCommitSync())
        records.forEach(record -> process(record));
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
    shutdownLatch.countDown();
  }
}
```

C/C++ (*librdkafka*):

```
void consume_loop(rd_kafka_t *rk,
                  rd_kafka_topic_partition_list_t *topics) {
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }

  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage && !rd_kafka_commit_message(rk, rkmessage, 0)) {
      msg_process(rkmessage);
      rd_kafka_message_destroy(rkmessage);
    }
  }

  err = rd_kafka_consumer_close(rk);
  if (err)
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

Python:

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
```

```
                    raise KafkaException(msg.error())
            else:
                consumer.commit(async=False)
                msg_process(msg)

    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

Go:

```
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        err = consumer.CommitMessage(e)
        if err == nil {
            msg_process(e)
        }

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

C#:

```
consumer.OnMessage += (_, msg) =>
{
    var err = consumer.CommitAsync().Result.Error;
    if (!err)
    {
        processMessage(msg);
    }
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}

while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

Для простоты в примере *rd_kafka_commit_message* используется перед обработкой сообщения, так как фиксация каждого сообщения на практике приводит к большим накладным расходам. Поэтому лучшим подходом является сбор пакета сообщений, выполнение синхронного коммита и затем после успешной фиксации

обработка сообщений.

Important: Правильное управление смещением имеет решающее значение, поскольку оно влияет на семантику доставки

2.5.4 Асинхронные коммиты

Каждый вызов commit API приводит к отправке брокеру запроса на фиксацию смещения. При использовании синхронного API потребитель блокируется до тех пор, пока запрос не будет успешно возвращен. Это может снизить общую пропускную способность, поскольку в противном случае потребитель мог бы обрабатывать записи, ожидающие фиксации. Одним из способов решения этой проблемы является увеличение объема данных, возвращаемых в каждом *poll()*, через параметр конфигурации *fetch.min.bytes*. Тогда брокер удерживает выборку до тех пор, пока не будет достигнуто достаточное количество данных (или не истечет срок *fetch.max.wait.ms*). Побочный эффект заключается в том, что способ также увеличивает количество дубликатов, с которыми приходится сталкиваться при случае сбоя.

Второй вариант – использовать асинхронные коммиты. Потребитель может отправить запрос и, не дожидаясь завершения запроса, немедленно вернуться.

```
public void run() {
  try {
    consumer.subscribe(topics);

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      consumer.commitAsync();
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
    shutdownLatch.countDown();
  }
}
```

C/C++ (*librdkafka*):

```
void consume_loop(rd_kafka_t *rk,
                  rd_kafka_topic_partition_list_t *topics) {
  static const int MIN_COMMIT_COUNT = 1000;

  int msg_count = 0;
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }

  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage) {
      msg_process(rkmessage);
```

```
      rd_kafka_message_destroy(rkmessage);

      if ((++msg_count % MIN_COMMIT_COUNT) == 0)
        rd_kafka_commit(rk, NULL, 1);
    }
  }

  err = rd_kafka_consumer_close(rk);
  if (err)
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

Единственное различие между этим примером и предыдущим заключается в том, что в вызове *rd_kafka_commit* включена асинхронная фиксация.

Изменения в **Python** очень похожи. Параметр *async* для *commit()* изменен на *True*. В примере значение передается явно, но асинхронная фиксация используется по умолчанию, если параметр не включен:

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(async=True)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

В **Go** для асинхронной фиксации необходимо выполнить коммит в goroutine:

```
msg_count := 0
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0 {
            go func() {
                offsets, err := consumer.Commit()
            }()
        }
```

```
        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

В **C#** для асинхронной фиксации необходимо вызвать метод *CommitAsync*:

```
var msgCount = 0;

consumer.OnMessage += (_, msg) =>
{
    processMessage(msg);
    msgCount += 1;
    if (msgCount % MIN_COMMIT_COUNT == 0)
    {
        consumer.CommitAsync();
    }
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}

while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

Поскольку такой способ помогает производительности, почему бы всегда не использовать асинхронные коммиты? Основная причина заключается в том, что потребитель не повторяет запрос в случае сбоя фиксации. Это то, что *commitSync* предлагает даром; он повторяется бесконечно, пока фиксация не будет выполнена, или не будет найдена неисправимая ошибка. Проблема с асинхронными коммитами связана с порядком фиксации – к тому времени, когда потребитель узнает, что фиксация не удалась, возможно, уже будет обработан следующий пакет сообщений, и даже будет отправлен следующий коммит. В этом случае повторная попытка старой фиксации может привести к дублированию.

Вместо того, чтобы усложнять свойства потребителей в попытках самостоятельного решения этой проблемы, API выдает обратный запрос при свершении коммита – и успешного, и при неудаче. При желании можно использовать этот обратный запрос для повторной фиксации, но тогда приходится сталкиваться с проблемой переназначения.

```
public void run() {
  try {
    consumer.subscribe(topics);
```

```
    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception␣
↪exception) {
          if (e != null)
            log.debug("Commit failed for offsets {}", offsets, e);
          }
      });
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
    shutdownLatch.countDown();
  }
}
```

Аналогичная функция доступна в **C/C++** (*librdkafka*), но ее необходимо настроить при инициализации:

```
static void on_commit(rd_kafka_t *rk,
                      rd_kafka_resp_err_t err,
                      rd_kafka_topic_partition_list_t *offsets,
                      void *opaque) {
  if (err)
    fprintf(stderr, "%% Failed to commit offsets: %s\n", rd_kafka_err2str(err));
}

void init_rd_kafka() {
  rd_kafka_conf_t *conf = rd_kafka_conf_new();
  rd_kafka_conf_set_offset_commit_cb(conf, on_commit);

  // initialization omitted
}
```

Аналогично, в **Python** обратный запрос может быть вызван любым коммитом и может быть передан в качестве параметра конфигурации конструктора потребителя:

```
from confluent_kafka import Consumer

def commit_completed(err, partitions):
    if err:
        print(str(err))
    else:
        print("Committed partition offsets: " + str(partitions))

conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'group.id': "foo",
        'default.topic.config': {'auto.offset.reset': 'smallest'},
        'on_commit': commit_completed}

consumer = Consumer(conf)
```

В **C#** можно использовать *Task*:

```
var msgCount = 0;

consumer.OnMessage += (_, msg) =>
{
    processMessage(msg);
    msgCount += 1;
    if (msgCount % MIN_COMMIT_COUNT == 0)
    {
        consumer.CommitAsync().ContinueWith(
            commitResult =>
            {
                if (commitResult.Error)
                {
                    Console.Error.WriteLine(commitResult.Error);
                }
                else
                {
                    Console.WriteLine(
                        $"Committed Offsets [{string.Join(", ", commitResult.Offsets)}]");
                }
            }
        )
    }
}
```

В **Go** события перебалансировки отображаются как события, возвращаемые методом *Poll()*. Для того чтобы увидеть эти события, необходимо создать потребителя с конфигурацией *go.application.rebalance.enable* и обработать события *AssignedPartitions* и *RevokedPartitions*, явно вызвав *Assign()* и *Unassign()* для *AssignedPartitions* и *RevokedPartitions* соответственно:

```
consumer, err := kafka.NewConsumer(&kafka.ConfigMap{
     "bootstrap.servers":    "host1:9092,host2:9092",
     "group.id":             "foo",
     "go.application.rebalance.enable": true})

msg_count := 0
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case kafka.AssignedPartitions:
        fmt.Fprintf(os.Stderr, "%% %v\n", e)
        c.Assign(e.Partitions)
    case kafka.RevokedPartitions:
        fmt.Fprintf(os.Stderr, "%% %v\n", e)
        c.Unassign()
    case *kafka.Message:
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0 {
            consumer.Commit()
        }

        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
```

```
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

Сбои фиксации смещения досаждают, когда последующие коммиты успешны, так как фактически они не должны приводить к повторным чтениям. Однако, если последняя фиксация завершается неудачно до того, как происходит перебалансировка или отключение потребителя, смещения сбрасываются до последнего коммита, и вероятнее всего, отображаются дубликаты. Поэтому общая схема заключается в том, чтобы объединить асинхронные коммиты в цикле опроса с синхронизированными коммитами при перебалансировках или отключении. Фиксация при отключении несложна, но необходимо найти способ закрепления поведения при перебалансировке. Для этого представленный ранее метод *subscribe()* имеет вариант, принимающий *ConsumerRebalanceListener*, который имеет два метода закрепления поведения перебалансировки.

В следующем примере синхронные фиксации включаются при перебалансировках и при отключении:

```
private void doCommitSync() {
  try {
    consumer.commitSync();
  } catch (WakeupException e) {
    // we're shutting down, but finish the commit first and then
    // rethrow the exception so that the main loop can exit
    doCommitSync();
    throw e;
  } catch (CommitFailedException e) {
    // the commit failed with an unrecoverable error. if there is any
    // internal state which depended on the commit, you can clean it
    // up here. otherwise it's reasonable to ignore the error and go on
    log.debug("Commit failed", e);
  }
}

public void run() {
  try {
    consumer.subscribe(topics, new ConsumerRebalanceListener() {
      @Override
      public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        doCommitSync();
      }

      @Override
      public void onPartitionsAssigned(Collection<TopicPartition> partitions) {}
    });

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      consumer.commitAsync();
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    try {
      doCommitSync();
    } finally {
```

```
      consumer.close();
      shutdownLatch.countDown();
    }
  }
}
```

Каждая перебалансировка имеет две фазы: отзыв и назначение партиции. Отзыв партиции всегда вызывается перед перебалансировкой и является последним шансом фиксации смещения перед переназначением. Фаза назначения партиции всегда вызывается после перебалансировки и может использоваться для установки начальной позиции назначенных партиций. В этом случае отзыв используется для синхронной фиксации текущих смещений.

Как правило, асинхронные коммиты следует считать менее безопасными, чем синхронные, так как последовательные неудачи фиксации приводят к увеличению обработки дубликатов. Можно снизить эту опасность, добавив логику для обработки ошибок фиксации в обратном запросе или периодически смешивая с вызовами *commitSync()*, но не следует добавлять слишком много сложностей при отсутствии прямой необходимости. При синхронных коммитах можно повысить надежность, увеличив число партиций топика и количество потребителей в группе. А если необходимо максимизировать пропускную способность при готовности некоторого увеличения числа дубликатов, то асинхронные коммиты могут стать хорошим вариантом.

Довольно очевидный момент, но стоит отметить, что асинхронные коммиты имеют смысл только для “at least once” доставки сообщений. Чтобы получить “at most once”, прежде чем считывать сообщение необходимо знать, успешна ли фиксация. А это подразумевает синхронную фиксацию, за исключением случая наличия возможности “непрочтения” сообщения после того, как обнаружится, что фиксация не удалась.

2.5.5 Мониторинг групп

Kafka включает в себя утилиту администратора для просмотра статуса групп потребителей.

Для получения списка активных групп в кластере можно использовать утилиту **kafka-consumer-groups**, включенную в дистрибутив **Kafka**. В большом кластере это может занять некоторое время, поскольку она собирает список путем проверки каждого брокера.

```
bin/kafka-consumer-groups --bootstrap-server host:9092 --list
```

Утилита **kafka-consumer-groups** также может быть использована для сбора информации о текущей группе. Пример просмотра актуальных назначений для группы *foo*:

```
bin/kafka-consumer-groups --bootstrap-server host:9092 --describe --group foo
```

В случае вызова утилиты во время перебалансировки, команда сообщает об ошибке. Тогда необходимо повторить попытку, в результате которой отображаются назначения для всех членов в текущей группе.

Глава 3

Kafka Java Producer

Платформа **ADS** включает в себя Java producer, поставляемый вместе с **Kafka**.

В документе представлен общий обзор работы поставщика, введение в параметры конфигурации для настройки и примеры из каждой клиентской библиотеки.

3.1 Концепция

Поставщик **Kafka** концептуально намного проще, чем потребитель, так как у него нет необходимости в групповой координации. Его основная функция состоит в том, чтобы сопоставить каждое сообщение с партицией топика и отправить запрос лидеру соответствующей партиции. Первое выполняется с помощью partitioner – механизма, выбирающего партицию посредством хэш-функции, и гарантирующего, что все сообщения с одинаковым (непустым) ключом отправляются в одну и ту же партицию. Если ключ не указан, то партиция определяется циклически с целью обеспечения равномерного распределения по партициям топика.

В каждой партиции кластера **Kafka** есть лидер и набор реплик среди брокеров – все записи проходят через лидера партиции, а реплики синхронизируются посредством выборки из него. Когда лидер отключается или выходит из строя, следующий лидер выбирается из числа синхронизированных реплик. В зависимости от того, как настроен поставщик, каждый запрос лидеру партиции может удерживаться до тех пор, пока реплики не одобрят запись. Это дает поставщику некий контроль над эксплуатацией сообщения при условии некоторой стоимости общей пропускной способности.

Сообщения, направленные лидеру партиции, не могут сразу считываться потребителями независимо от настроек подтверждения поставщика. Только когда все синхронизированные реплики подтверждают запись, сообщение считается зафиксированным, что делает его доступным для чтения. Такой подход гарантирует, что уже прочитанные сообщения не могут быть потеряны по причине сбоя брокера. Но это также подразумевает, что сообщения, подтвержденые только лидером (то есть *acks=1*), могут быть потеряны в случае, если лидер партиции терпит неудачу до момента копирования сообщений репликами. Тем не менее, в большинстве случаев на практике такой способ часто является разумным компромиссом для обеспечения жизнеспособности сообщений (durability), при этом не оказывая существенного влияния на пропускную способность.

Большая часть тонкостей вокруг поставщиков связана с достижением высокой пропускной способности с учетом пакетирования/сжатия и обеспечением гарантий доставки сообщений. Далее приведены наиболее распространенные параметры настройки поведения поставщиков.

3.2 Конфигурация

Полный список параметров конфигурации доступен в документе Настройки платформы Arenadata Streaming. Но некоторые из ключевых параметров и их влияние на поведение поставщиков описаны в текущей

главе.

3.2.1 Базовая конфигурация (Core Configuration)

Для того, чтобы поставщик мог найти кластер **Kafka**, необходимо установить свойство *bootstrap.servers*. Так же, хотя это и не требуется обязательно, но всегда следует устанавливать *client.id*, поскольку это позволяет легко сопоставлять запросы в брокере со сделавшим их инстансом клиента. Данные настройки одинаковы для клиентов **Java**, **C/C++**, **Python**, **Go** и **.NET**.

3.2.2 Жизнеспособность сообщений (Message Durability)

Жизнеспособность сообщений, записанных в **Kafka**, можно контролировать с помощью параметра *acks*. Значение по умолчанию *1* требует от лидера партиции явного подтверждения об успешно выполненной записи. Самая сильная гарантия, которую предоставляет **Kafka** – *acks=all* – сообщение не только допущено к записи лидером партиции, но и успешно скопировано на все синхронизированные реплики. Можно также использовать значение *0* для максимизации пропускной способности, но тогда отсутствует гарантия успешной записи сообщения в журнал брокера, так как в этом случае брокер не отправляет ответ, что также означает невозможность определения смещение сообщения. Для клиентов **C/C++**, **Python**, **Go** и **.NET** это является конфигурацией для каждого отдельного топика, но ее можно применять глобально с помощью вложенной конфигурации *default_topic_conf* в **C/C++** и *default.topic.config* в **Python**, **Go** и **.NET**.

3.2.3 Порядок сообщений (Message Ordering)

Как правило, сообщения записываются в брокер в том же порядке, в котором они поступают от клиента поставщика. Однако если разрешить повторные попытки сообщений, установив для них значение больше *0* (*0* – значение по умолчанию), может измениться их порядок, так как повтор возможен только после свершения успешной записи. Чтобы избежать переупорядочения, можно установить параметр *max.in.flight.requests.per.connection* в значение *1*, тогда брокеру одновременно может быть отправлен только один запрос. В случае без подключения повторных попыток сообщений брокер сохраняет порядок получаемых записей, но при этом могут быть пробелы из-за отдельных сбоев отправки.

3.2.4 Пакетирование и сжатие (Batching/Compression)

Поставщики **Kafka** пакетируют отправляемые сообщения для повышения пропускной способности. С клиентом **Java** можно управлять максимальным размером каждого пакета сообщений в параметре *batch.size*. Для большего времени на заполнение пакетов доступен параметр *linger.ms*, на значение которого поставщик задерживает отправку. Установкой *compression.type* включается сжатие, оно охватывает полные пакеты сообщений, поэтому большие пакеты обычно означают более высокую степень сжатия.

С клиентами **C/C++**, **Python**, **Go** и **.NET** для установки ограничения на количество сообщений в пакете используется *batch.num.messages*, сжатие включается с помощью *compression.codec*.

3.2.5 Ограничения очереди (Queuing Limits)

Ограничение общего объема доступной Java-клиенту памяти для сбора неотправленных сообщений контролируется параметром *buffer.memory*. При достижении установленного предела поставщик блокирует последующий набор записей до тех пор, пока *max.block.ms* не выводит исключение. Кроме того, чтобы избежать бесконечного хранения сообщений, можно установить тайм-аут в *request.timeout.ms*. Если это время ожидания истекает до того, как сообщение может быть успешно отправлено, то оно удаляется из очереди и генерируется исключение.

Клиенты **C/C++**, **Python**, **Go** и **.NET** имеют аналогичные настройки. Параметр *queue.buffering.max.messages* – для ограничения общего количества сообщений, поставляемых в очередь в любой момент времени (для отчетов о передаче, повторных попытках или доставке). И параметр *queue.buffering.max.ms* – для ограничения периода времени ожидания клиентом заполнения пакета перед отправкой брокеру.

3.3 Примеры

3.3.1 Начальная настройка

Поставщик **Java** создается с помощью стандартного файла свойств *Properties*:

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("bootstrap.servers", "host1:9092,host2:9092");
config.put("acks", "all");
new KafkProducer<K, V>(config);
```

Ошибки конфигурации приводят к появлению *KafkaException* от конструктора *KafkaProducer*. Основное отличие *librdkafka* заключается в том, что она обрабатывает ошибки для каждого параметра напрямую:

```
char hostname[128];
char errstr[512];

rd_kafka_conf_t *conf = rd_kafka_conf_new();

if (gethostname(hostname, sizeof(hostname))) {
  fprintf(stderr, "%% Failed to lookup hostname\n");
  exit(1);
}

if (rd_kafka_conf_set(conf, "client.id", hostname,
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
  fprintf(stderr, "%% %s\n", errstr);
  exit(1);
}

if (rd_kafka_conf_set(conf, "bootstrap.servers", "host1:9092,host2:9092",
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
  fprintf(stderr, "%% %s\n", errstr);
  exit(1);
}

rd_kafka_topic_conf_t *topic_conf = rd_kafka_topic_conf_new();

if (rd_kafka_topic_conf_set(topic_conf, "acks", "all",
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
  fprintf(stderr, "%% %s\n", errstr);
  exit(1);
}

/* Create Kafka producer handle */
rd_kafka_t *rk;
if (!(rk = rd_kafka_new(RD_KAFKA_PRODUCER, conf,
                        errstr, sizeof(errstr)))) {
  fprintf(stderr, "%% Failed to create new producer: %s\n", errstr);
  exit(1);
}
```

В **Python**:

```
from confluent_kafka import Producer
import socket
```

```
conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'client.id': socket.gethostname(),
        'default.topic.config': {'acks': 'all'}}

producer = Producer(conf)
```

В **Go**:

```
import (
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

p, err := kafka.NewProducer(&kafka.ConfigMap{
    "bootstrap.servers": "host1:9092,host2:9092",
    "client.id": socket.gethostname(),
    "default.topic.config": kafka.ConfigMap{'acks': 'all'}
})

if err != nil {
    fmt.Printf("Failed to create producer: %s\n", err)
    os.Exit(1)
}
```

В **C#**:

```
using Confluent.Kafka;
using System.Net;

...

var config = new Dictionary<string, object>
{
    { "bootstrap.servers", "host1:9092,host2:9092" },
    { "client.id", Dns.GetHostName() },
    { "default.topic.config", new Dictionary<string, object>
        {
            { "acks", "all" }
        }
    }
}

using (var producer = new Producer<Null, string>(config, null, new StringSerializer(Encoding.
↪UTF8)))
{
    ...
}
```

3.3.2 Асинхронные записи

Все записи являются асинхронными по умолчанию. Поставщик **Java** включает в себя API *send()*, возвращающий future, которое можно опрашивать для получения результата отправки:

```
final ProducerRecord<K, V> = new ProducerRecord<>(topic, key, value);
Future<RecordMetadata> future = producer.send(record);
```

В *librdkafka* сначала необходимо создать дескриптор *rd_kafka_topic_t* для топика, в который планируется запись, а затем использовать *rd_kafka_produce* для отправки в него сообщений. Например:

```
rd_kafka_topic_t *rkt = rd_kafka_topic_new(rk, topic, topic_conf);

if (rd_kafka_produce(rkt, RD_KAFKA_PARTITION_UA,
                     RD_KAFKA_MSG_F_COPY,
                     payload, payload_len,
                     key, key_len,
                     NULL) == -1) {
  fprintf(stderr, "%% Failed to produce to topic %s: %s\n",
       topic, rd_kafka_err2str(rd_kafka_errno2err(errno)));
}
```

Конкретную топику конфигурацию можно назначить третьему аргументу *rd_kafka_topic_new* – тогда необходимо передать *topic_conf* и добавить настройку для подтверждений. Значение *NULL* приводит к использованию поставщиком конфигурации по умолчанию.

Второй аргумент для *rd_kafka_produce* может использоваться для установки желаемой партиции для сообщения. При установленом значении *RD_KAFKA_PARTITION_UA*, как в данном примере, выбор партиции для сообщения осуществляется механизмом partitioner по умолчанию. Третий аргумент указывает, что *librdkafka* должна скопировать информацию и ключ, что позволяет освободить его по возвращении.

В **Python** отправка инициируется методом *produce* с передачей значения и по необходимости – ключа, партиции и обратного вызова. Запрос возвращается немедленно без значения:

```
producer.produce(topic, key="key", value="value")
```

Аналогично, в **Go** отправка инициируется методом *Produce()* с передачей объекта *Message‘ object and an optional ‘‘chan Event*, применяемого для прослушивания результата отправки. Объект *Message* содержит непрозрачное поле *interface{}*, которое может использоваться для передачи произвольных данных вместе с сообщением последующему обработчику событий.

```
delivery_chan := make(chan kafka.Event, 10000)
err = p.Produce(&kafka.Message{
    TopicPartition: kafka.TopicPartition{Topic: "topic", Partition: kafka.PartitionAny},
    Value: []byte(value)},
    delivery_chan,
)
```

В **C#** отправка инициируется вызовом метода *ProduceAsync* в инстансе *Producer*. Например:

```
producer.ProduceAsync("topic", key, value);
```

При необходимости применения некоторого кода после завершения записи может быть предоставлен обратный вызов. В **Java** это реализовано как объект *Callback*:

```
final ProducerRecord<K, V> = new ProducerRecord<>(topic, key, value);
producer.send(record, new Callback() {
  public void onCompletion(RecordMetadata metadata, Exception e) {
    if (e != null)
      log.debug("Send failed for record {}", record, e);
  }
});
```

В реализации **Java** следует избегать дорогостоящей работы с обратным вызовом, поскольку он выполняется в потоке ввода-вывода поставщика.

Аналогичная функция доступна в *librdkafka*, но ее необходимо настраивать при инициализации:

```
static void on_delivery(rd_kafka_t *rk,
                        const rd_kafka_message_t *rkmessage
                        void *opaque) {
  if (rkmessage->err)
    fprintf(stderr, "%% Message delivery failed: %s\n",
            rd_kafka_message_errstr(rkmessage));
}

void init_rd_kafka() {
  rd_kafka_conf_t *conf = rd_kafka_conf_new();
  rd_kafka_conf_set_dr_msg_cb(conf, on_delivery);

  // initialization ommitted
}
```

Обратный вызов доставки (delivery callback) в *librdkafka* осуществляется в потоке пользователя путем вызова *rd_kafka_poll*. Распространенным шаблоном является вызов функции после каждого вызова API поставщика, но этого может быть недостаточно для обеспечения регулярных отчетов о доставке, если скорость создания сообщений не равномерна. Так же данный API не предоставляет прямого способа блокировки для получения результата доставки конкретного сообщения. При наличии такой необходимости рекомендуется рассмотреть пример синхронной записи (*Синхронные записи*).

В **Python** параметр *callback* можно передать с помощью любого вызываемого средства, например, лямбды, функции, связанного метода или вызываемого объекта. Хотя метод *produce()* сразу ставит сообщение в очередь для пакетной обработки, сжатия и передачи брокеру, он не свершает обработку каких-либо событий (то есть подтверждений и обратных вызовов, которые они инициируют) до вызова *poll()*.

```
def acked(err, msg):
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err))
    else
        print("Message produced: %s" % (str(msg))

producer.produce(topic, key="key", value="value", callback=acked)

# Wait up to 1 second for events. Callbacks will be invoked during
# this method call if the message is acknowledged.
producer.poll(1)
```

В **Go** можно использовать канал отчета о доставке в *Produce*, чтобы дождаться результата отправки сообщения:

```
e := <-delivery_chan
m := e.(*kafka.Message)

if m.TopicPartition.Error != nil {
    fmt.Printf("Delivery failed: %v\n", m.TopicPartition.Error)
} else {
    fmt.Printf("Delivered message to topic %s [%d] at offset %v\n",
               *m.TopicPartition.Topic, m.TopicPartition.Partition, m.TopicPartition.Offset)
}

close(delivery_chan)
```

В **C#** есть два варианта. Первый: можно использовать *ProduceAsync*, возвращающий стандартный объект *Task*, который выполняет await (приостановку выполнения метода до завершения выполнения ожидаемой задачи), обрабатывается с помощью метода *.ContinueWith* или ожидает использования методов *.Wait* или

.WaitAll:

```
var deliveryReportTask = producer.ProduceAsync("topic", key, val);
deliveryReportTask.ContinueWith(task =>
{
    Console.WriteLine($"Partition: {task.Result.Partition}, Offset: {task.Result.Offset}");
});
```

Во втором варианте используется *.ProduceAsync*, который принимает реализацию *IDeliveryHandler*. Данный подход следует использовать при необходимости получения уведомлений о доставке сообщений (или сбое доставки) строго в порядке подтверждения брокером, поскольку *Tasks* могут выполняться в любом потоке, что не гарантирует их упорядоченность.

3.3.3 Синхронные записи

Чтобы сделать запись синхронной, следует дождаться возвращения future. Как правило, это плохая затея, так как она может существенно снизить пропускную способность, но в некоторых случаях может быть оправдана.

```
Future<RecordMetadata> future = producer.send(record);
RecordMetadata metadata = future.get();
```

Аналогичная возможность может быть достигнута в **C/C++** и **Python** с помощью обратного вызова доставки, но это более трудоемко. Полный пример приведен по ссылке. Клиент **Python** также содержит метод *flush()*, имеющий тот же эффект:

```
producer.produce(topic, key="key", value="value")
producer.flush()
```

В **Go** осуществляется через канал доставки, посредством вызова метода *Produce()*:

```
delivery_chan := make(chan kafka.Event, 10000)
err = p.Produce(&kafka.Message{
    TopicPartition: kafka.TopicPartition{Topic: "topic", Partition: kafka.PartitionAny},
    Value: []byte(value)},
    delivery_chan
)

 e := <-delivery_chan
 m := e.(*kafka.Message)
```

Для ожидания подтверждения всех сообщений используется метод *Flush()*:

```
p.Flush()
```

Важно обратить внимание, что *Flush()* обслуживает только канал *Events()* поставщика, а не каналы доставки, указанные приложением. Если *Flush()* вызывается, и при этом никакая горутина не обрабатывает канал доставки, то буфер может заполниться и привести к истечению времени ожидания.

В **C#** необходимо получить доступ к свойству *.Result* объекта *Task*, возвращенного из *.ProduceAsync*, которое будет блокироваться до тех пор, пока не станет доступен отчет о доставке:

```
var deliveryReport = producer.ProduceAsync("topic", key, value).Result;
```

Глава 4

Kafka Connect

Kafka Connect – компонент **Apache Kafka** с открытым исходным кодом, является основой для подключения **Kafka** к внешним системам, таким как базы данных, хранилища key-value, поисковые индексы и файловые системы. С **Kafka Connect** можно использовать существующие реализации коннекторов для перемещения данных в сервис **Kafka** и из него:

- *Source Connector* – принимает базы данных и обновляет таблицы потоков для топиков **Kafka**. Также собирает метрики со всех серверов приложений в топики **Kafka**, делая данные доступными для потоковой обработки с низкой задержкой;
- *Sink Connector* – доставляет данные из топиков **Kafka** во вторичные индексы, такие как **Elasticsearch**, или в пакетные системы для автономного анализа, такие как **Hadoop**.

Kafka Connect ориентирован на потоковую передачу данных из сервиса **Kafka** и в него, что упрощает написание высококачественных, надежных и высокопроизводительных плагинов. Это также позволяет фреймворку давать гарантии, которые трудно достичь с помощью других структур. **Kafka Connect** является неотъемлемым компонентом конвейера **ETL** в сочетании с сервисом **Kafka** и потоковой обработкой.

Kafka Connect может работать либо как автономный процесс для выполнения заданий на одной машине (например, сбор журналов), либо как распределенный, масштабируемый, отказоустойчивый сервис, поддерживающий всю структуру. Это позволяет сократить масштаб до разработки, тестирования и небольших продуктовых развертываний с низким барьером для входа и низкими эксплуатационными накладными расходами, а также увеличить масштаб поддержки конвейера данных большой организации.

Основные преимущества использования **Kafka Connect**:

- *Data Centric Pipeline* – использование значимых абстракций данных для извлечения или передачи данных в **Kafka**;
- *Flexibility and Scalability (гибкость и масштабируемость)* – работа с потоковыми и пакетно-ориентированными системами на одном узле или масштабирование до сервиса по всей ширине организации;
- *Reusability and Extensibility (повторное использование и расширяемость)* – использование существующих коннекторов и возможность расширения их для адаптации к конкретным потребностям и сокращения времени на разработку.

4.1 Connectors & Tasks

Копирование данных между сервисом **Kafka** и сторонней системой осуществляется посредством создаваемых пользователями инстансов *Kafka Connectors*. Коннекторы бывают двух видов: *SourceConnectors* – импортируют данные из другой системы, и *SinkConnectors* – экспортируют данные в другую систему. Например,

JDBCSourceConnector импортирует реляционную базу данных в **Kafka**, а *HDFSSinkConnector* экспортирует содержимое топика **Kafka** в файлы **HDFS**.

Реализации класса Connector не выполняют копирование данных самостоятельно: их конфигурация описывает набор данных для копирования, и Connector отвечает за разбиение этого задания на набор задач – Tasks, которые могут быть распределены между объектами **Kafka Connect**. Tasks также бывают двух видов: SourceTask и SinkTask. При необходимости реализация класса Connector может отслеживать изменения данных внешних систем и запрашивать реконфигурацию задачи.

С назначением данных, которые должны быть скопированы, каждая задача *Task* должна скопировать свое подмножество данных в сервис **Kafka** или из него. Данные, которые копирует коннектор, должны быть представлены как партиционированный поток, аналогично модели топика **Kafka**, где каждая партиция представляет собой упорядоченную последовательность записей со смещениями. Каждой задаче назначается подмножество партиций для обработки. Порой это сопоставление очевидно: каждый файл в наборе файлов журнала можно считать партицией, каждую строку в файле – записью, а смещения – просто позициии в файле. В иных случаях сопоставление с моделью требует больше усилий: коннектор **JDBC** может сопоставить каждую таблицу с партицией, но смещение менее ясно. Один из возможных вариантов сопоставления это использовать в качестве смещения последнюю запрашиваемую отметку времени при генерации запросов.

Source Connector, создавший две задачи, которые копируют данные из входных партиций и записывают в сервис **Kafka**, приведен на Рис.4.1..

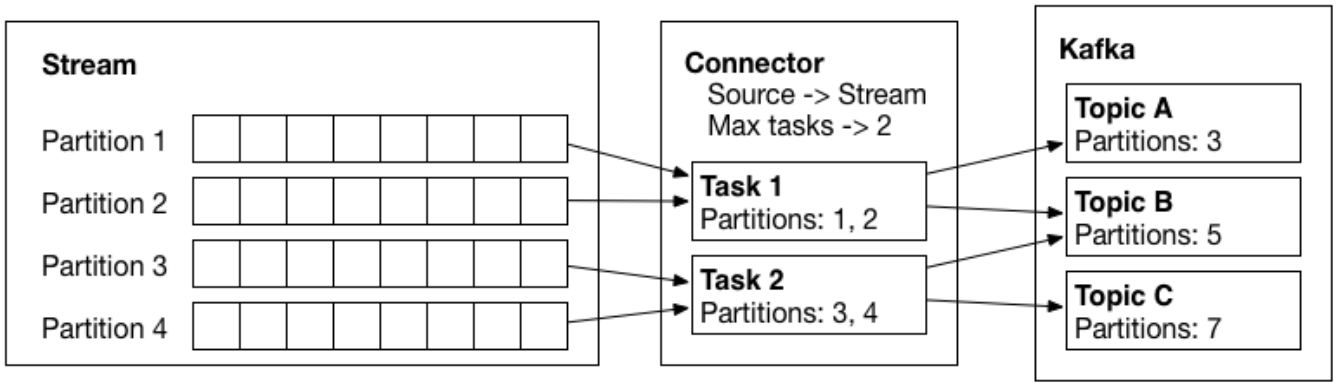


Рис.4.1.: Пример реализации *Source Connector*

4.2 Partitions & Records

Каждая партиция представляет собой упорядоченную последовательность записей ключ-значение, где и ключи, и значения могут иметь сложные структуры. Поддерживаются многие примитивные типы, а также массивы, структуры и вложенные структуры данных. Для большинства типов можно напрямую использовать стандартные типы **Java**, такие как *java.lang.Integer*, *java.lang.Map* и *java.lang.Collection*. Для структурированных записей следует использовать класс *Struct*.

На Рис.4.2. представлен партиционированный поток: модель данных, в которой коннекторы сопоставляют все системы *source* и *sink*. Каждая запись содержит ключи и значения (со схемами), идентификатор партиции и смещения в ней.

Для отслеживания структуры и совместимости записей в партициях схемы (*Schemas*) могут быть включены в каждую запись. Поскольку схемы обычно генерируются “на лету” на основе источника данных, класс *SchemaBuilder* включен, что делает их построение очень простым.

Schemas Определение абстрактного типа данных. Типы данных могут быть примитивными типами (целочисленные типы, типы с плавающей запятой, логические, строки и байты) или сложными

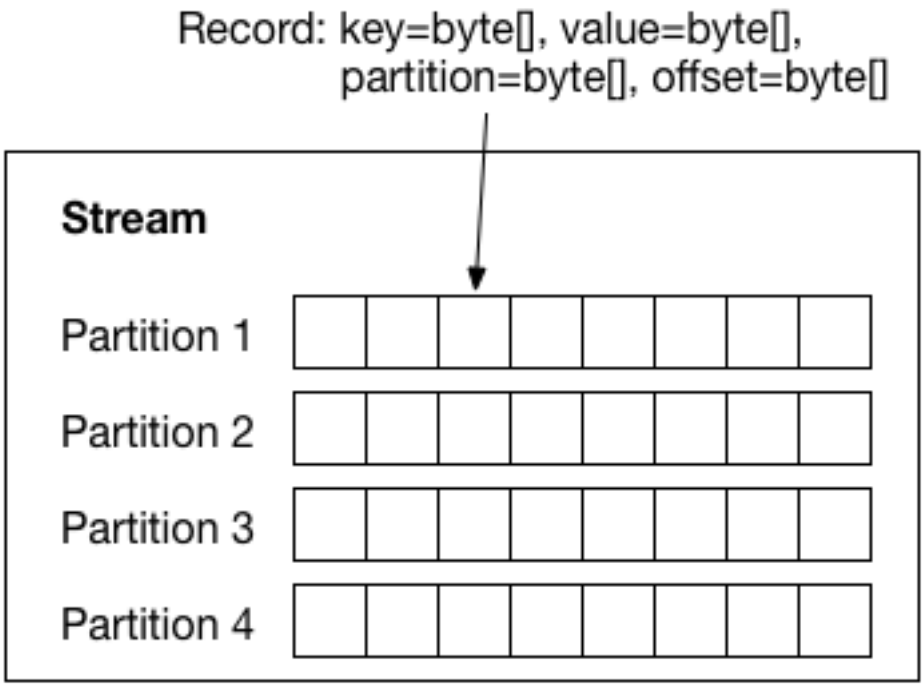


Рис.4.2.: Пример партиционированного потока

типами (типизированные массивы, карты с одной схемой ключей и схемами значений, а также структурами, которые имеют фиксированный набор имен полей, каждый из которых имеет схема связанных значений). Любой тип может быть указан как необязательный, что позволяет его опускать (в результате чего значения отсутствуют) и может указывать значение по умолчанию.

Такой формат данных среды выполнения не предполагает какого-либо конкретного формата сериализации; это преобразование осуществляется с помощью *Converter*, которые обрабатывают формат времени выполнения *org.apache.kafka.connect.data* и сериализованные данные *byte[]*.

Converter Интерфейс конвертера обеспечивает поддержку перевода между форматом данных выполнения Kafka Connect и byte[]. Внутренне это включает промежуточный шаг к формату, используемому слоем сериализации (например, JsonNode, GenericRecord, Message).

В дополнение к ключу и значению записи имеют идентификаторы партиций и смещения, которые используются фреймворком для периодической фиксации смещений обработанных данных. В случае сбоя обработка может возобновиться с последнего зафиксированного смещения, что позволяет избежать повторной обработки и дублирования событий.