

**Exhibit 2 to
Intellectual Ventures I LLC and Intellectual Ventures II LLC
Amended Infringement Contentions**

**Amended Infringement Claim Chart for
U.S. Patent No. 8,407,722 (“the ’722 Patent”)**

The Accused Systems and Services include, without limitation, Comerica’s systems that utilize Kafka; all past, current, and future systems and services that operate in the same or substantially similar manner as the specifically identified systems and services; and all past, current and future Comerica’s systems and services that have the same or substantially similar features as the specifically identified systems and services (“Comerica Count I Systems and Services”).¹ Upon information and belief, Comerica’s systems and services directly infringe the asserted claims of the ’722 Patent through at least using Kafka in its various systems and offerings. For example, as shown in the source code revisions below, Comerica uses Kafka for various applications. In addition, Comerica directly infringes the ’722 Patent via testing the various systems and offerings. On information and belief, Comerica also indirectly infringes the ’722 Patent through inducement. On information and belief, Comerica also contributes to the direct infringement through its vendors and other third parties’ uses of Comerica’s Systems and Services that utilize Kafka by providing components relating to Kafka that have no substantial non-infringing uses. On information and belief, Comerica with knowledge at least at the time of filing the complaint, supplies its customers, employees, and/or vendors and other third parties with access to its systems and services through various client offerings and/or internal services that allows its customers, employees, and/or vendors and other third parties to use the claimed method. On information and belief, Comerica servers, or servers that are run on behalf of Comerica, are involved and/or perform the method of the asserted claims of the ’722 Patent. On information and belief, Comerica provides instructions on how to use its various offerings and the use of which results in infringement of the ’722 Patent claims through performance of the claimed methods.

Pursuant to P.R. 3-1(b), IV identifies Kafka as used in Comerica’s systems and offerings as the Accused Instrumentality. Information regarding how specifically this Kafka is used for what product and offerings is non-public. IV intends to obtain discovery regarding how the Kafka is used in Comerica products and services and will update its infringement positions once this discovery is completed. For example, IV served a 30(b)(6) deposition notice on Comerica on June 26, 2024, seeking information relating to Comerica’s use and implementation of Kafka, including use and implementation of Kafka to support services that are offered to

¹ For the avoidance of doubt, Plaintiffs do not accuse the public clouds of Defendant, to the extent those services are provided by a cloud provider with a license to Plaintiffs’ patents that covers Defendant’s activities. [REDACTED]

IV accuses Comerica private clouds that implement Kafka and non-licensed public clouds that Comerica uses to support Kafka for its systems and services. To the extent any of these licenses are relevant to Defendant’s activities, Plaintiffs will meet and confer with Defendants about the impact of such license(s).

[REDACTED]

Comerica's customers and/or employees.² This information requested by IV includes for example technical documents that show the operation of Comerica's use and implementation of Kafka, Comerica source code relating to Kafka, whether third parties have information relating to Comerica's use and implementation of Kafka, and other technical information. See Plaintiffs' Initial 30(b)(6) Notice to Defendant (Comerica) at 9-10. IV served an amended 30(b)(6) notice on August 16, 2024, requesting additional relevant information. See generally Plaintiffs' Amended 30(b)(6) Notice to Defendant (Comerica). IV has also served 30(b)(1) deposition notices for several technical Comerica employees that likely possess knowledge of Comerica's use and implementation of Kafka. To date, Comerica has not offered any deposition dates for either potential corporate witnesses or witnesses having personal knowledge of Comerica's use and implementation of Kafka. These amended contentions are based on information available to IV as of the date of service of these contentions, and are served without the benefit of even a single technical deposition to understand how Comerica uses and implements Kafka in its systems and services. Further, fact discovery remains open and Plaintiffs intend to serve additional discovery to better understand Comerica's use and implementation of Kafka. Accordingly, Plaintiffs reserve all rights to amend these contentions consistent with the local rules. Further, IV has filed a motion to compel seeking discovery including the identification of Comerica software applications that utilize Kafka, and reserve the right to supplement these contentions once the Court has resolved this pending dispute.

² IV understands that Comerica alleges that it does not implement Spark and/or Kafka. IV has requested additional information regarding Comerica's use of Spark and/or Kafka, *see* J. An E-mail Dated July 25, 2024, but has not received a response as of service of these contentions. As described in these contentions, discovery is pending and IV has yet to take Comerica's 30(b)(6) deposition or depositions of its employees with technical knowledge as to Spark and/or Kafka, and according reserves its rights to seek information as to how Comerica uses (or has used) Spark and/or Kafka.

U.S. Patent No. 8,407,722	
Claim 14	Comerica Count II Systems and Services
14[pre] A method comprising: 14[a][i] providing, using a processing device of an input source,	Upon information and belief, Comerica performs “providing, using a processing device of an input source.” Comerica systems utilize Kafka, a distributed streaming system that runs as a cluster of one or more servers. Using the producer client and API (e.g. providing), a first entity/service/application (e.g. processing device) associated with a data server or database sends streams of records or events to the respective Kafka topic(s). <u>How does Kafka work in a nutshell?</u> Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments. Servers: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss. Clients: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala including the higher-level Kafka Streams library, for Go, Python, C/C++, and many other programming languages as well as REST APIs. See https://kafka.apache.org/documentation/ (last accessed on November 10, 2023).

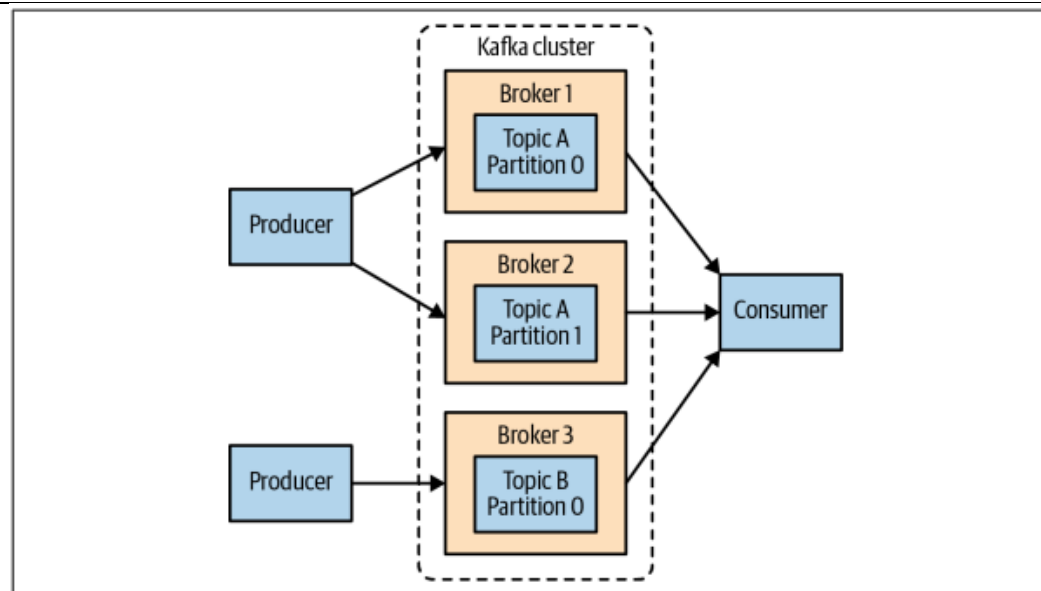
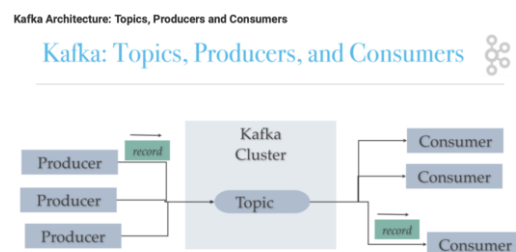


Figure 2-2. A simple Kafka cluster

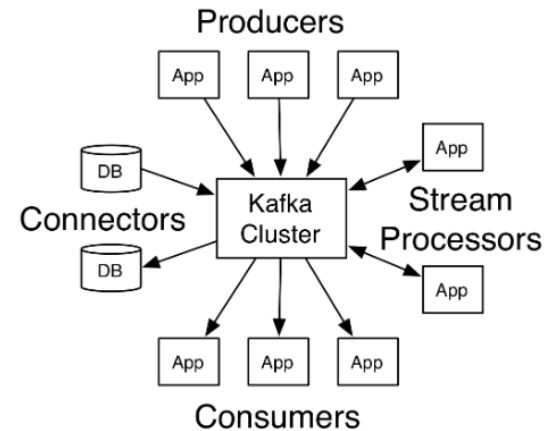
See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.



See <http://cloudurable.com/blog/kafka-architecture/index.html> (last accessed March 15, 2024).

Kafka has four core APIs:

- The [Producer API](#) allows an application to publish a stream of records to one or more Kafka topics.
- The [Consumer API](#) allows an application to subscribe to one or more topics and process the stream of records produced to them.
- The [Streams API](#) allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The [Connector API](#) allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.



See <https://kafka.apache.org/20/documentation.html> (last accessed March 15, 2024).

An **event** is any type of action, incident, or change that's identified or recorded by software or applications. For example, a payment, a website click, or a temperature reading, along with a description of what happened.

In other words, an event is a combination of notification—the element of when-ness that can be used to trigger some other activity—and state. That state is usually fairly small, say less than a megabyte or so, and is normally represented in some structured format, say in JSON or an object serialized with Apache Avro™ or Protocol Buffers.

See <https://developer.confluent.io/what-is-apache-kafka/> (last accessed August 26, 2024).

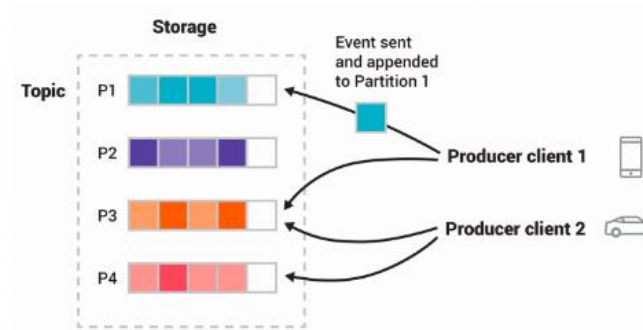
For example, the Producer API allows a first entity/service/application to send streams of records or events to Kafka (e.g., providing, using a processing device of an input source).

2.1 Producer API

The Producer API allows applications to send streams of data to topics in the Kafka cluster.

See <https://kafka.apache.org/documentation/> (last accessed on November 10, 2023).

Producers are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various [guarantees](#) such as the ability to process events exactly-once.



See <https://kafka.apache.org/intro> (last accessed on November 13, 2023).

The `KafkaProducer` class and the `ProducerRecord` class work together to enable transmitting messages within the network with the help of the brokers. They allow records or events to have an associated topic and partition to where they belong in addition to having data content with an identifiable key.



	<pre>public class KafkaProducer<K,V> extends Object implements Producer<K,V> A Kafka client that publishes records to the Kafka cluster. The producer is <i>thread safe</i> and sharing a single producer instance across threads will generally be faster than having multiple instances. Here is a simple example of using the producer to send records with strings containing sequential numbers as the key/value pairs. Properties props = new Properties(); props.put("bootstrap.servers", "localhost:9092"); props.put("acks", "all"); props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer"); props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer"); Producer<String, String> producer = new KafkaProducer<>(props); for (int i = 0; i < 100; i++) producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i), Integer.toSt producer.close(); The producer consists of a pool of buffer space that holds records that haven't yet been transmitted to the server as well as a background I/O thread that is responsible for turning these records into requests and transmitting them to the cluster. Failure to close the producer after use will leak these resources. The <code>send()</code> method is asynchronous. When called it adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency.</pre>
	See: https://kafka.apache.org/23/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html (last accessed March 11, 2024). The below figure represents a <code>ProducerRecord</code> class with the parameters topic, partition, key, and value.



ProducerRecord

```
public ProducerRecord(String topic,
    Integer partition,
    K key,
    V value)
```

Creates a record to be sent to a specified topic and partition

Parameters:
topic - The topic the record will be appended to
partition - The partition to which the record should be sent
key - The key that will be included in the record
value - The record contents

```
public class ProducerRecord<K,V>
    extends Object
```

A key/value pair to be sent to Kafka. This consists of a topic name to which the record is being sent, an optional partition number, and an optional key and value.

If a valid partition number is specified that partition will be used when sending the record. If no partition is specified but a key is present a partition will be chosen using a hash of the key. If neither key nor partition is present a partition will be assigned in a round-robin fashion.

The record also has an associated timestamp. If the user did not provide a timestamp, the producer will stamp the record with its current time. The timestamp eventually used by Kafka depends on the timestamp type configured for the topic.

- If the topic is configured to use CreateTime, the timestamp in the producer record will be used by the broker.
- If the topic is configured to use LogAppendTime, the timestamp in the producer record will be overwritten by the broker with the broker local time when it appends the message to its log.

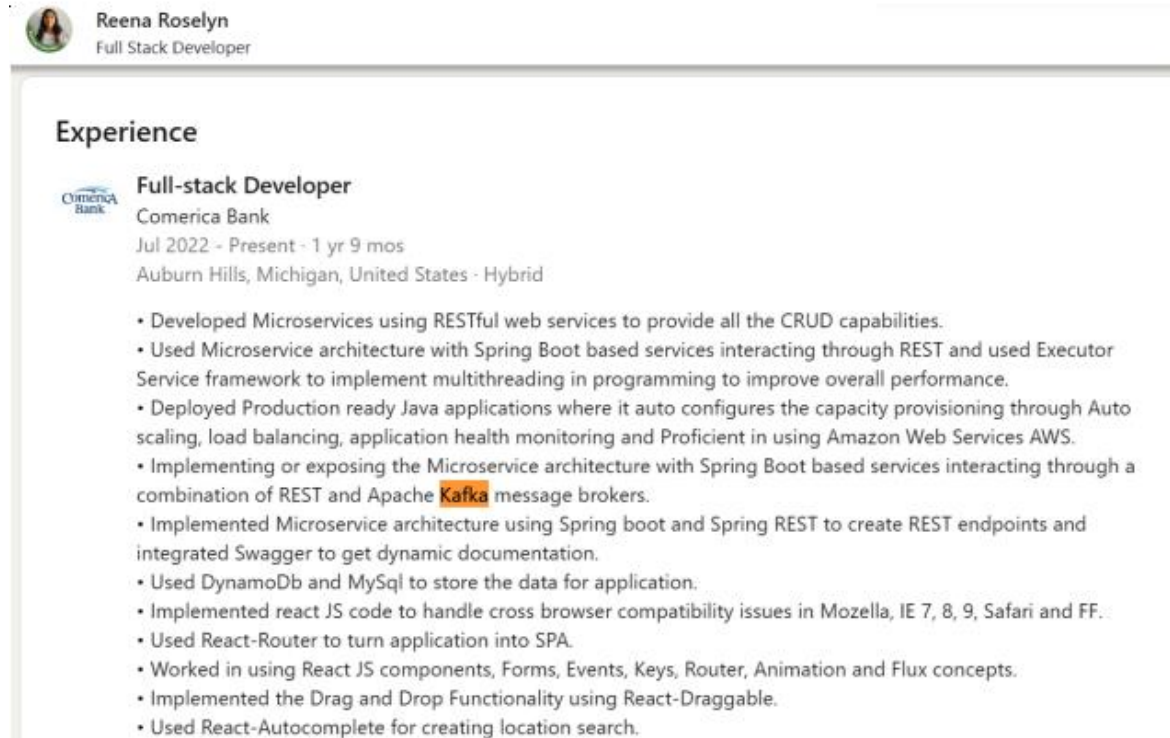
In either of the cases above, the timestamp that has actually been used will be returned to user in `RecordMetadata`

See <https://kafka.apache.org/0102/javadoc/org/apache/kafka/clients/producer/ProducerRecord.html> (last accessed March 15, 2024).

Comerica implements Kafka to provide end users with updates in real time.

See <https://www.jobzmall.com/comerica-bank/job/mobile-application-developer-7> (last accessed on March 19, 2024).

See <https://www.jobzmall.com/comerica-bank/job/principal-data-engineer-17> (last accessed on March 19, 2024).



Reena Roselyn
Full Stack Developer

Experience

Full-stack Developer
Comerica Bank
Jul 2022 - Present · 1 yr 9 mos
Auburn Hills, Michigan, United States · Hybrid

- Developed Microservices using RESTful web services to provide all the CRUD capabilities.
- Used Microservice architecture with Spring Boot based services interacting through REST and used Executor Service framework to implement multithreading in programming to improve overall performance.
- Deployed Production ready Java applications where it auto configures the capacity provisioning through Auto scaling, load balancing, application health monitoring and Proficient in using Amazon Web Services AWS.
- Implementing or exposing the Microservice architecture with Spring Boot based services interacting through a combination of REST and Apache **Kafka** message brokers.
- Implemented Microservice architecture using Spring boot and Spring REST to create REST endpoints and integrated Swagger to get dynamic documentation.
- Used DynamoDb and MySQL to store the data for application.
- Implemented react JS code to handle cross browser compatibility issues in Mozella, IE 7, 8, 9, Safari and FF.
- Used React-Router to turn application into SPA.
- Worked in using React JS components, Forms, Events, Keys, Router, Animation and Flux concepts.
- Implemented the Drag and Drop Functionality using React-Draggable.
- Used React-Autocomplete for creating location search.

See <https://www.linkedin.com/in/reena-roselyn-365988149/> (last accessed on March 19, 2024).

See <https://apps.apple.com/us/app/comerica-mobile-banking/id403598968> (last accessed on November 11, 2023).



See <https://apps.apple.com/us/app/comerica-treasury-mobile/id1051410838?platform=iphone> (last accessed on November 11, 2023).

As described in the source code production below, librdkafka is the Apache Kakfa C/C++ client library.

librdkafka

The Apache Kafka C/C++ client library

Main Page

Related Pages

Data Structures

Files

librdkafka

librdkafka documentation

Introduction

Configuration

Statistics

C API

C++ API

Introduction to librdkafka - the Apache Ka

Configuration properties

Statistics

Deprecated List

Data Structures

Files

librdkafka documentation

librdkafka is the Apache Kafka C/C++ client library.

Introduction

For an introduction and manual to librdkafka see [Introduction to librdkafka - the Apache Kafka C/C++ client library](#)

Configuration

librdkafka is highly configurable to meet any deployment demands. It is usually safe to leave most configuration properties to their default values.

See [Configuration properties](#) for the full list of supported configuration properties.

Remarks

Application developers are recommended to provide a non-hardcoded interface to librdkafka's string based name-value configuration property interface, allowing users to configure any librdkafka property directly without alterations to the application. This allows for seamless upgrades where linking to a new version of librdkafka automatically provides new configuration based features.

See <https://docs.confluent.io/platform/current/clients/librdkafka/html/index.html> (last accessed August 27, 2024).

Source Code Evidence

10

IPR2025-00200 / 722 Patent

Page 11 of 65

EX1018

[REDACTED]

	[REDACTED]
14[a][ii] a data representation to a client device, different from the input source, coupled to a routing network,	Upon information and belief, Comerica's systems comprises "a data representation to a client device, different from the input source, coupled to a routing network." Kafka is a distributed streaming system that is run as a cluster of one or more servers. Using the producer client and API, a first entity/service/application sends streams of records or events (e.g. data representation) to the respective Kafka topic. As an example, these records or events include dynamic objects such as transactions. They are routed to a destination entity/service/application (e.g. client device) consumer client with the consumer API via brokers in the network (e.g. coupled to a routing network).



	Kafka is a distributed <i>event streaming platform</i> that lets you read, write, store, and process <i>events</i> (also called <i>records</i> or <i>messages</i> in the documentation) across many machines. Example events are payment transactions, geolocation updates from mobile phones, shipping orders, sensor measurements from IoT devices or medical equipment, and much more. These events are organized and stored in <i>topics</i>. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. See https://kafka.apache.org/quickstart (last accessed on November 13, 2023).
--	--

How does Kafka work in a nutshell?

Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance [TCP network protocol](#). It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

Servers: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run [Kafka Connect](#) to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

Clients: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by [dozens of clients](#) provided by the Kafka community: clients are available for Java and Scala including the higher-level [Kafka Streams](#) library, for Go, Python, C/C++, and many other programming languages as well as REST APIs.

See <https://kafka.apache.org/documentation/> (last accessed on November 10, 2023).

Producers are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various [guarantees](#) such as the ability to process events exactly-once.

See <https://kafka.apache.org/intro> (last accessed on November 13, 2023).

2.2 Consumer API

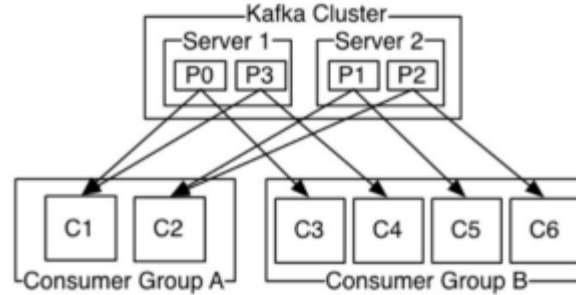
The Consumer API allows applications to read streams of data from topics in the Kafka cluster.

Examples showing how to use the consumer are given in the [javadocs](#).

To use the consumer, you can use the following maven dependency:

See <https://kafka.apache.org/documentation.html> (last accessed March 15, 2024).

For example, destination entities/services/applications (e.g. client device) can initialize a Kafka consumer instance to connect to the Kafka cluster. For all consumer instances reading from the same topic, they are grouped with distinctive IDs. An even distributed assignment of partitions to the topics will happen across each consumer group. The consumer API offers the functionality of offset committing to provide an accurate consumption of the messages in a topic stream.



See <https://www.confluent.io/blog/a-comprehensive-rest-proxy-for-kafka/> (last accessed March 15, 2024)

Usage Examples

The consumer APIs offer flexibility to cover a variety of consumption use cases. Here are some examples to demonstrate how to use them.

Automatic Offset Committing

This example demonstrates a simple usage of Kafka's consumer api that relying on automatic offset committing.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
}
```

The connection to the cluster is bootstrapped by specifying a list of one or more brokers to contact using the configuration `bootstrap.servers`. This list is just used to discover the rest of the brokers in the cluster and need not be an exhaustive list of servers in the cluster (though you may want to specify more than one in case there are servers down when the client is connecting).

Setting `enable.auto.commit` means that offsets are committed automatically with a frequency controlled by the config `auto.commit.interval.ms`.

In this example the consumer is subscribing to the topics `foo` and `bar` as part of a group of consumers called `test` as configured with `group.id`.

[REDACTED]

	See https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html (last accessed March 15, 2024). For example, Kafka producer and consumer client applications communicate in a cluster through the Kafka brokers via a network (e.g. routing network) to read or write from streams of events. A Kafka client communicates with the Kafka brokers via the network for writing (or reading) events. Once received, the brokers will store the events in a durable and fault-tolerant manner for as long as you need—even forever. See https://kafka.apache.org/quickstart (last accessed July 26, 2024). Source Code Evidence [REDACTED]
14[a][iii] wherein the data representation includes at least one live object	Upon information and belief, Comerica’s systems are configured such that “wherein the data representation includes at least one live object recognizable by the client device.” For example, Kafka is a distributed streaming system that is run as a cluster of one or more servers. Using the producer client and API, a first entity/service/application associated with a data server sends streams of records or events (e.g. data

recognizable by the client device,	representation) to the respective Kafka topic. When a producer reads or writes to Kafka, they are writing an event(s) that consists of something that has occurred (e.g. at least one live object). Each new event/record updates a previous event/record within a topic. Consumer entity/service/application (e.g. client device) that subscribes to the said topic will receive the updated event occurrence. Kafka is a distributed <i>event streaming platform</i> that lets you read, write, store, and process events (also called <i>records</i> or <i>messages</i> in the documentation) across many machines. Example events are payment transactions, geolocation updates from mobile phones, shipping orders, sensor measurements from IoT devices or medical equipment, and much more. These events are organized and stored in topics. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. See https://kafka.apache.org/quickstart (last accessed on November 13, 2023). An event records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event: • Event key: "Alice" • Event value: "Made a payment of \$200 to Bob" • Event timestamp: "Jun. 25, 2020 at 2:06 p.m."
------------------------------------	---

Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

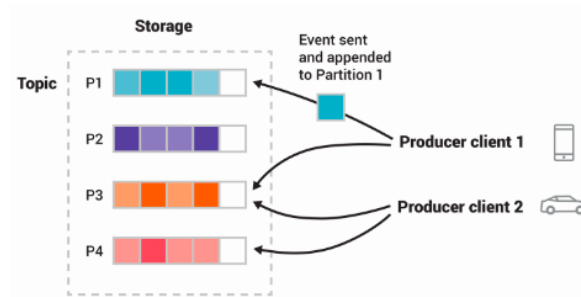


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

See <https://kafka.apache.org/intro> (last accessed on November 13, 2023).

For example, destination entities/services/applications (e.g. client device) can initialize a Kafka consumer instance to connect to the Kafka cluster. For all consumer instances reading from the same topic, they are grouped with distinctive IDs. An even assignment of partitions of the topics will be distributed across each consumer group. The consumer API offers the functionality of offset committing to provide an accurate consumption of the messages in a topic stream.

Each consumer in a group can dynamically set the list of topics it wants to subscribe to through one of the `subscribe` APIs. Kafka will deliver each message in the subscribed topics to one process in each consumer group. This is achieved by balancing the partitions between all members in the consumer group so that each partition is assigned to exactly one consumer in the group. So if there is a topic with four partitions, and a consumer group with two processes, each process would consume from two partitions.

See <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html> (last accessed March 15, 2024).

Usage Examples

The consumer APIs offer flexibility to cover a variety of consumption use cases. Here are some examples to demonstrate how to use them.

Automatic Offset Committing

This example demonstrates a simple usage of Kafka's consumer api that relying on automatic offset committing.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
}
```

The connection to the cluster is bootstrapped by specifying a list of one or more brokers to contact using the configuration `bootstrap.servers`. This list is just used to discover the rest of the brokers in the cluster and need not be an exhaustive list of servers in the cluster (though you may want to specify more than one in case there are servers down when the client is connecting).

Setting `enable.auto.commit` means that offsets are committed automatically with a frequency controlled by the config `auto.commit.interval.ms`.

In this example the consumer is subscribing to the topics `foo` and `bar` as part of a group of consumers called `test` as configured with `group.id`.

Id.

Source Code Evidence

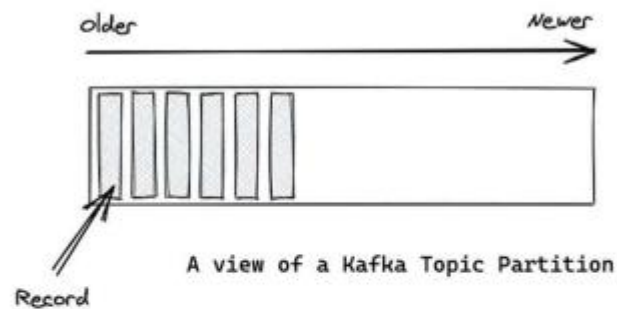
[REDACTED]

	[REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
14[a][iv] and causing the client device to respond to the live object of the data representation by determining an object identifier (ID) of the live object	Upon information and belief, Comerica's systems comprises "causing the client device to respond to the live object of the data representation by determining an object identifier (ID) of the live object." When a producer writes an event(s) to Kafka, they are written to topics. A topic is a timeline of events, where each contains an event key (e.g., object identifier (ID) of the live object) and the topic can be divided into several partitions. Kafka facilitates the receipt of events where destination entity/service/applications (e.g. client device(s)) subscribed to that topic can read through the stream and consume the events (e.g. respond to the live object). An event records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event: • Event key: "Alice" • Event value: "Made a payment of \$200 to Bob" • Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

See <https://kafka.apache.org/intro> (last accessed on November 13, 2023)

Partitions

Kafka's topics are divided into several partitions. While the topic is a logical concept in Kafka, a partition is the smallest storage unit that *holds a subset of records owned by a topic*. Each partition is a single log file where records are written to it in an append-only fashion.



When talking about the content inside a partition, I will use the terms **record** and **message** interchangeably.

Offsets and the ordering of messages

The records in the partitions are each assigned a sequential identifier called the offset, which is unique for each record within the partition.

The offset is an incremental and immutable number, maintained by Kafka. When a record is written to a partition, it is appended to the end of the log, assigning the next sequential offset. Offsets are particularly useful for consumers when reading records from a partition. We'll come to that at a later point.

See <https://medium.com/event-driven-utopia/understanding-kafka-topic-partitions-ae40f80552e8> (last accessed March 11, 2024).

For example, Kafka destination entities/services/applications (e.g. client device) initializing a consumer instance subscribing to topics are noted as subscribers. Consumers subscribing to topics will be grouped to keep track, and the consumer API offers the functionality of offset assignments to provide recovery in case of any disconnects or failures.

org.apache.kafka.clients.consumer

Class KafkaConsumer<K,V>

java.lang.Object

org.apache.kafka.clients.consumer.KafkaConsumer<K,V>

See <https://kafka.apache.org/22/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html> (last accessed March 15, 2024).

Each consumer in a group can dynamically set the list of topics it wants to subscribe to through one of the `subscribe` APIs. Kafka will deliver each message in the subscribed topics to one process in each consumer group. This is achieved by balancing the partitions between all members in the consumer group so that each partition is assigned to exactly one consumer in the group. So if there is a topic with four partitions, and a consumer group with two processes, each process would consume from two partitions.

Id.

called *subscribers* or *readers*. The consumer *subscribes* to one or more topics and reads the messages in the order in which they were produced to each partition. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. The *offset*—an integer value that continually increases—is another piece of metadata that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset, and the following message has a greater offset (though not necessarily monotonically greater). By storing the next possible offset for each partition, typically in Kafka itself, a consumer can stop and restart without losing its place.

See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

	Source Code Evidence [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
14[a][v] and to register for updates of the live object with the routing network, such that registering the client device with the routing network provides client connection information to a node in the routing network; and	Upon information and belief, Comerica's systems comprises a “register for updates of the live object with the routing network, such that registering the client device with the routing network provides client connection information to a node in the routing network.” A consumer can subscribe to a topic, or topics, (e.g. register for updates of the live object) and Kafka brokers (e.g. node in the routing network) facilitate the receipt of those events to consumer entity/service/applications user devices as they read through the topic stream. Kafka uses a network protocol such as TCP, to facilitate message requests and delivery through the Kafka brokers. A Kafka client communicates with the Kafka brokers via the network for writing (or reading) events. Once received, the brokers will store the events in a durable and fault-tolerant manner for as long as you need—even forever. See https://kafka.apache.org/quickstart (last accessed July 26, 2024). Each consumer in a group can dynamically set the list of topics it wants to subscribe to through one of the <code>subscribe</code> APIs. Kafka will deliver each message in the subscribed topics to one process in each consumer group. This is achieved by balancing the partitions between all members in the consumer group so that each partition is assigned to exactly one consumer in the group. So if there is a topic with four partitions, and a consumer group with two processes, each process would consume from two partitions.

See <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html> (last accessed March 15, 2024).

Kafka uses a binary protocol over TCP. The protocol defines all apis as request response message pairs. All messages are size delimited and are made up of the following primitive types.

The client initiates a socket connection and then writes a sequence of request messages and reads back the corresponding response message. No handshake is required on connection or disconnection. TCP is happier if you maintain persistent connections used for many requests to amortize the cost of the TCP handshake, but beyond this penalty connecting is pretty cheap.

The client will likely need to maintain a connection to multiple brokers, as data is partitioned and the clients will need to talk to the server that has their data. However it should not generally be necessary to maintain multiple connections to a single broker from a single client instance (i.e. connection pooling).

The server guarantees that on a single TCP connection, requests will be processed in the order they are sent and responses will return in that order as well. The broker's request processing allows only a single in-flight request per connection in order to guarantee this ordering. Note that clients can (and ideally should) use non-blocking IO to implement request pipelining and achieve higher throughput. i.e., clients can send requests even while awaiting responses for preceding requests since the outstanding requests will be buffered in the underlying OS socket buffer. All requests are initiated by the client, and result in a corresponding response message from the server except where noted.

The server has a configurable maximum limit on request size and any request that exceeds this limit will result in the socket being disconnected.

See https://kafka.apache.org/0100/protocol.html#protocol_network (last accessed July 29, 2024).

As an example, Kafka Streams applications read through a changelog stream or table of records with unique keys. When a new event is processed, the state of the table updates as well with a new column addition.

KTable is an abstraction of a *changelog stream* from a primary-keyed table. Each record in this changelog stream is an update on the primary-keyed table with the record key as the primary key.

See <https://kafka.apache.org/23/javadoc/org/apache/kafka/streams/kstream/KTable.html> (last accessed on November 10, 2023).

Let's illustrate this with an example. Imagine a table that tracks the total number of pageviews by user (first column of diagram below). Over time, whenever a new pageview event is processed, the state of the table is updated accordingly. Here, the state changes between different points in time – and different revisions of the table – can be represented as a changelog stream (second column).

See <https://docs.confluent.io/platform/current/streams/concepts.html> (last accessed on November 10, 2023)

As an example, Kafka utilizes DNS to facilitate connectivity between any producer or consumer entities/services/applications (e.g. client device). On information and belief, for example, a first connection made by a consumer will connect to a bootstrap server, where the broker will provide metadata regarding other details of the Kafka cluster. DNS resolution provides registering the client device with the routing network on a first connection (e.g. registering the client) which provides client connection information to a node in the routing network.

DNS Resolution for Kafka clients

DNS resolution is an important aspect of Kafka client connectivity as it enables the clients to connect to the correct Kafka brokers. When a Kafka client is configured to connect to a Kafka cluster, it typically specifies a list of broker hostnames or IP addresses in its configuration.

When a Kafka client is started, it first reads its configuration file to determine the initial list of bootstrap brokers. These brokers are specified as a comma-separated list of hostname and port pairs (e.g.,

```
broker1.example.com:9092,broker2.example.com:9092).
```

The Kafka client then performs DNS resolution to obtain the IP addresses of the specified hostnames. By default, the client uses the operating system's default DNS resolution behavior, which typically involves sending DNS queries to a DNS resolver provided by the operating system.

See <https://levelup.gitconnected.com/dns-resolution-for-kafka-clients-2e76b28e4717> (last accessed March 15, 2024).

Source Code Evidence

[illegible]

	[REDACTED]
14[b] sending, using the processing device of the input source, an update message to the routing network, wherein the update message identifies the live object and contains update data that updates a property of the live object,	Upon information and belief, Comerica’s systems perform “sending, using the processing device of the input source, an update message to the routing network, wherein the update message identifies the live object and contains update data that updates a property of the live object.” Each new event/record that is written to a Kafka topic from a first entity/service/application (e.g. sending, using the processing device of an input source) comprises content (e.g. update data) that includes an event key (e.g. update message identifies the live object). Using the <i>send</i> function, a record or event is asynchronously sent to its corresponding topic. The key for each event is used to determine the corresponding partition/topic it should be written to. ProducerRecord <pre>public ProducerRecord(String topic, Integer partition, K key, V value)</pre> Creates a record to be sent to a specified topic and partition Parameters: - topic - The topic the record will be appended to - partition - The partition to which the record should be sent - key - The key that will be included in the record - value - The record contents See https://kafka.apache.org/0102/javadoc/org/apache/kafka/clients/producer/ProducerRecord.html (last accessed March 15, 2024).

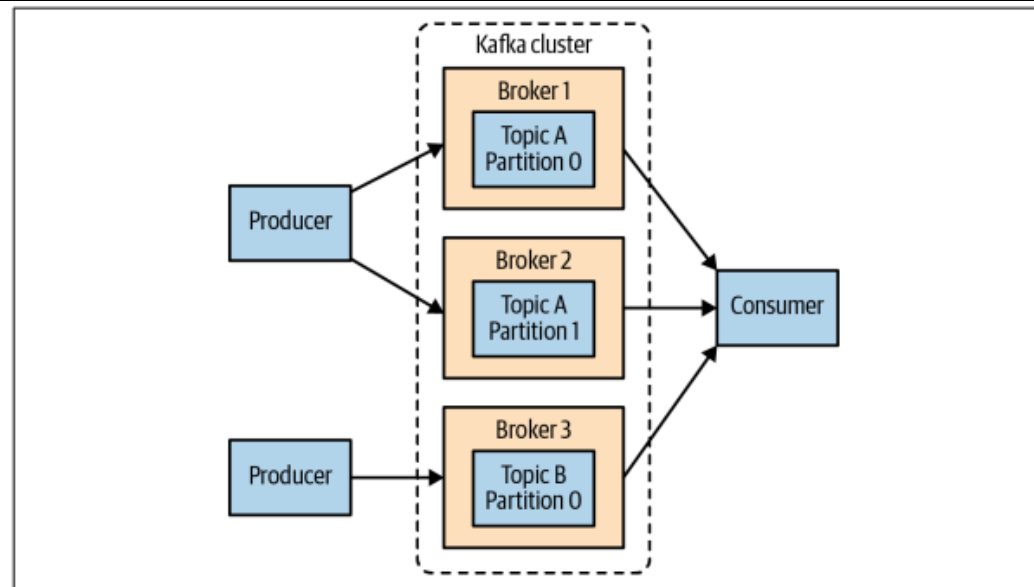
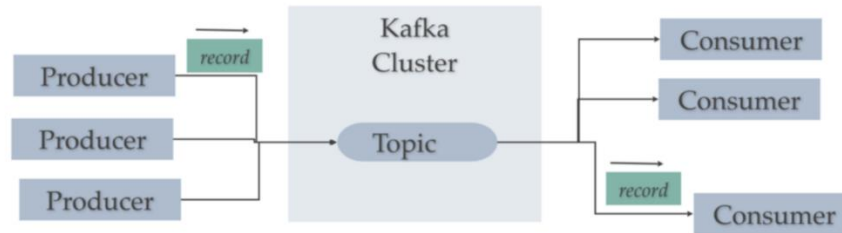


Figure 2-2. A simple Kafka cluster

See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

Kafka: Topics, Producers, and Consumers



See <http://cloudurable.com/blog/kafka-architecture/index.html> (last accessed March 15, 2024).

send

```
public Future<RecordMetadata> send(ProducerRecord<K,V> record)
```

Asynchronously send a record to a topic. Equivalent to `send(record, null)`. See `send(ProducerRecord, Callback)` for details.

Specified by:

`send` in interface `Producer<K,V>`

Parameters:

`record` - The record to send

Returns:

A future which will eventually contain the response information

See <https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html> (last accessed on November 10, 2023).

For example, records or events include dynamic objects, such as transactions, and make up respective topic streams that can be partitioned across multiple Kafka brokers as they are written by a first

entity/service/application (e.g. sending, using the processing device of the input source). For consumer clients subscribed to and reading these topic streams, the API provides continuous coordination and rebalancing on how these topics are partitioned.

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of \$200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

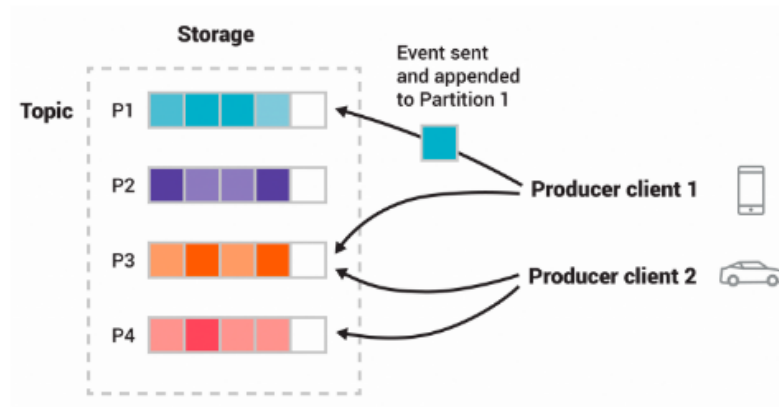


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

See <https://kafka.apache.org/intro> (last accessed on March 19, 2024).

At the heart of the consumer API is a simple loop for polling the server for more data. Once the consumer subscribes to topics, the poll loop handles all details of coordination, partition rebalances, heartbeats, and data fetching, leaving the developer with a clean API that simply returns available data from the assigned partitions. The main body of a consumer will look as follows:

```
try {
    while (true) { ❶
        ConsumerRecords<String, String> records = consumer.poll(100); ❷
        for (ConsumerRecord<String, String> record : records) ❸
        {
            log.debug("topic = %s, partition = %d, offset = %d,"
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());

            int updatedCount = 1;
            if (custCountryMap.containsKey(record.value())) {
                updatedCount = custCountryMap.get(record.value()) + 1;
            }
            custCountryMap.put(record.value(), updatedCount)

            JSONObject json = new JSONObject(custCountryMap);
            System.out.println(json.toString(4)) ❹
        }
    }
}
```

See <https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html> (last accessed March 15, 2024).

A **KTable** is an abstraction of a **changelog stream**, where each data record represents an update. More precisely, the value in a data record is interpreted as an “UPDATE” of the last value for the same record key, if any (if a corresponding key doesn’t exist yet, the update will be considered an INSERT). Using the table

See <https://docs.confluent.io/platform/current/streams/concepts.html> (last accessed March 15, 2024).

Source Code Evidence

[REDACTED]

	[REDACTED]
14[c] wherein a gateway device at the routing network is configured to identify a category of the update message based on the input source, to determine a node type to which the identified category maps, and to route the update message to the node, having the node type, at the routing network,	Upon information and belief, Comerica's systems include "wherein a gateway device at the routing network is configured to identify a category of the update message based on the input source, to determine a node type to which the identified category maps, and to route the update message to the node, having the node type, at the routing network." Comerica systems include Kafka Clusters that comprise multiple Kafka brokers (e.g. nodes), one of which is the cluster controller. It handles a new event written and read by the event key (e.g. identify a category of the update message) to the broker responsible. A broker is selected based on the topic/partition to which the event belongs (e.g. determine a node type to which the identified category maps).

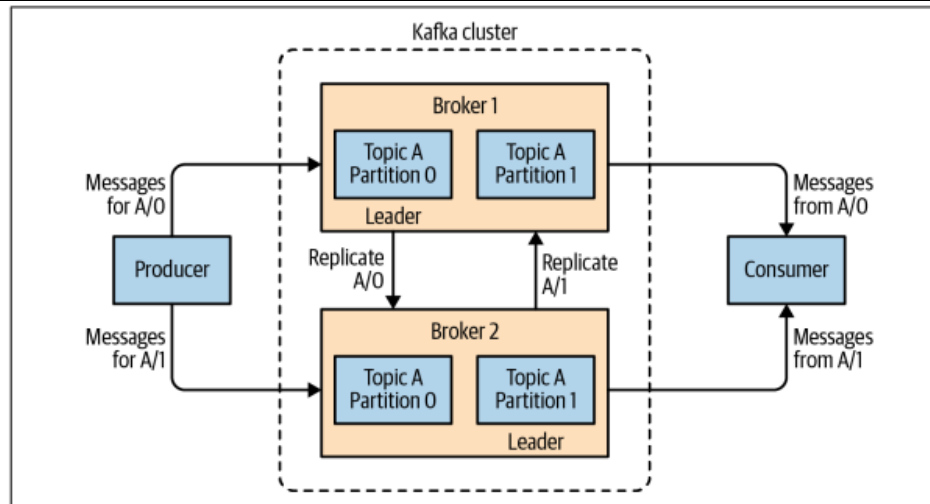


Figure 1-7. Replication of partitions in a cluster

A single Kafka server is called a *broker*. The broker receives messages from producers, assigns offsets to them, and writes the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been published. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.

Kafka brokers are designed to operate as part of a *cluster*. Within a cluster of brokers, one broker will also function as the cluster *controller* (elected automatically from the live members of the cluster). The controller is responsible for administrative operations, including assigning partitions to brokers and monitoring for broker failures. A partition is owned by a single broker in the cluster, and that broker is called the *leader* of the partition. A replicated partition (as seen in Figure 1-7) is assigned to additional brokers, called *followers* of the partition. Replication provides redundancy of messages in the partition, such that one of the followers can take over leadership if there is a broker failure. All producers must connect to the leader in order to publish messages, but consumers may fetch from either the leader or one of the followers. Cluster operations, including partition replication, are covered in detail in Chapter 7.

See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

Servers: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run [Kafka Connect](#) to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

For example, records or events include dynamic objects, such as transactions, and make up respective topic streams that can be partitioned across multiple Kafka brokers. As they are written by a first entity/service/application, any new updates will also be routed to the topic stream/partition it belongs to at respective Kafka brokers (e.g. route the update message to the node, have the node type).

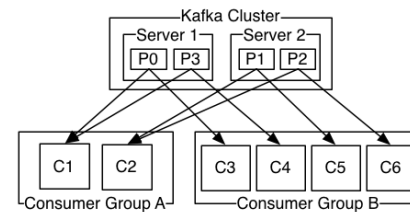
An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of \$200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

See <https://kafka.apache.org/documentation/> (last accessed on November 10, 2023).

A Kafka Cluster comprises multiple Kafka brokers (e.g., nodes). A broker includes multiple topics (and partitions). A broker is selected based on the topic/partition to which the event belongs. The broker stores the events inside the partitions of the topics.



A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups.
Consumer group A has two consumer instances and group B has four.

See <https://www.confluent.io/blog/a-comprehensive-rest-proxy-for-kafka/> (last accessed March 15, 2024).

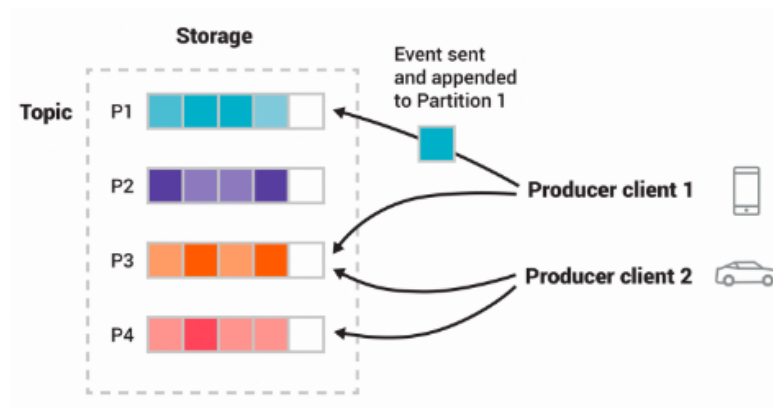


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

See <https://kafka.apache.org/documentation/> (last accessed on November 10, 2023).

For example, when messages are written on a topic, it adds to the end of that specified partition to keep the order in which the writes are happening. Furthermore, when a consumer group of multiple consumer clients is reading from this topic, it will read from the offset/index it previously had.

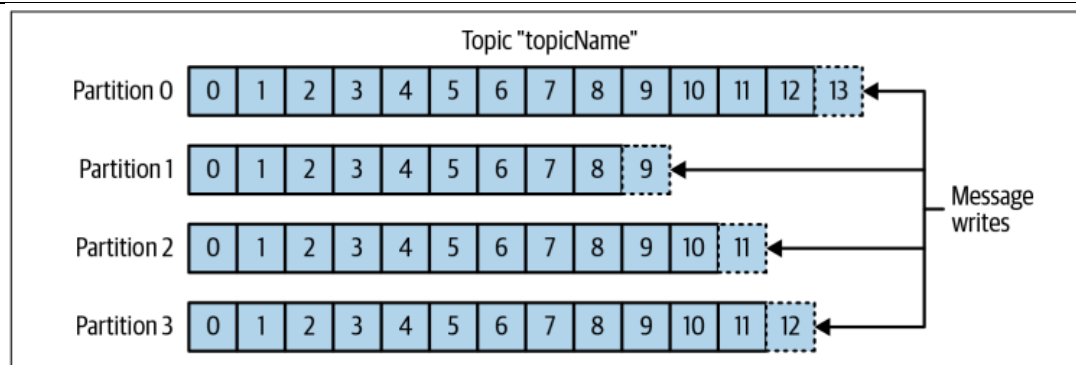


Figure 1-5. Representation of a topic with multiple partitions

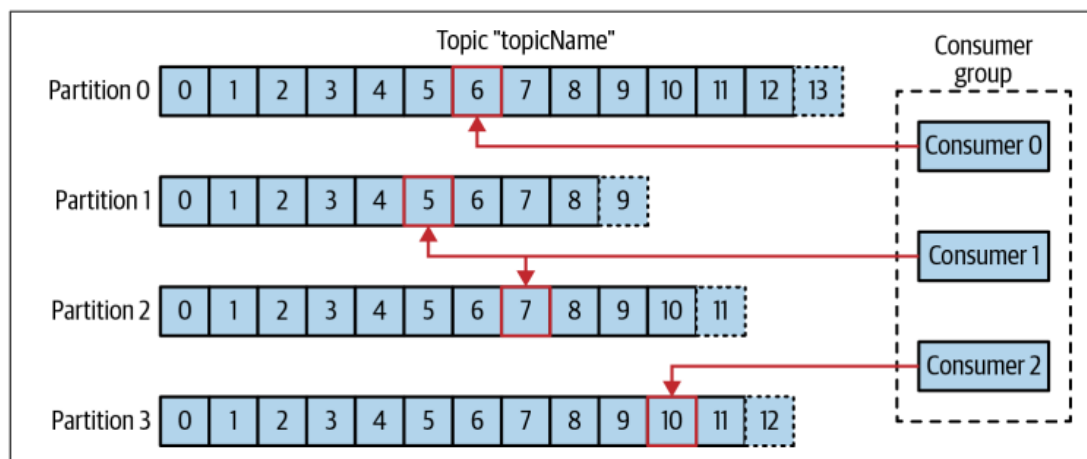


Figure 1-6. A consumer group reading from a topic

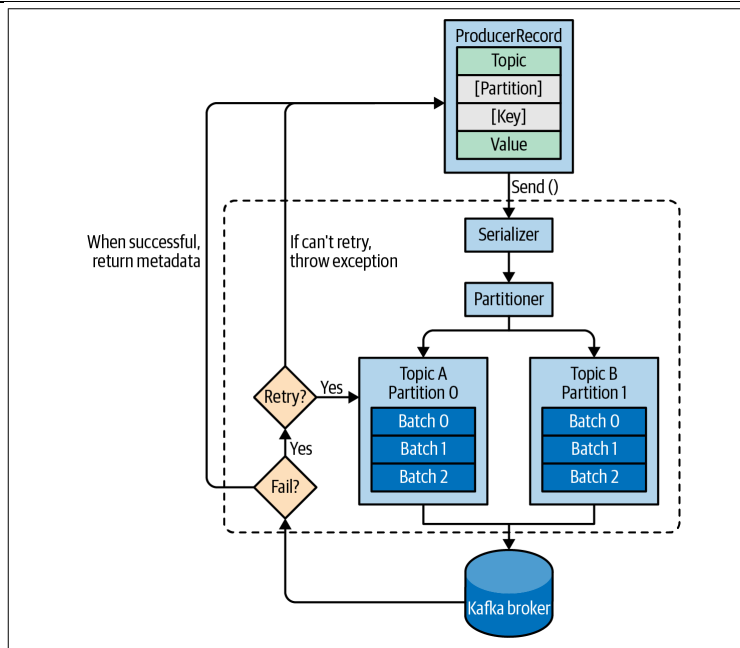


Figure 3-1. High-level overview of Kafka producer components

See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

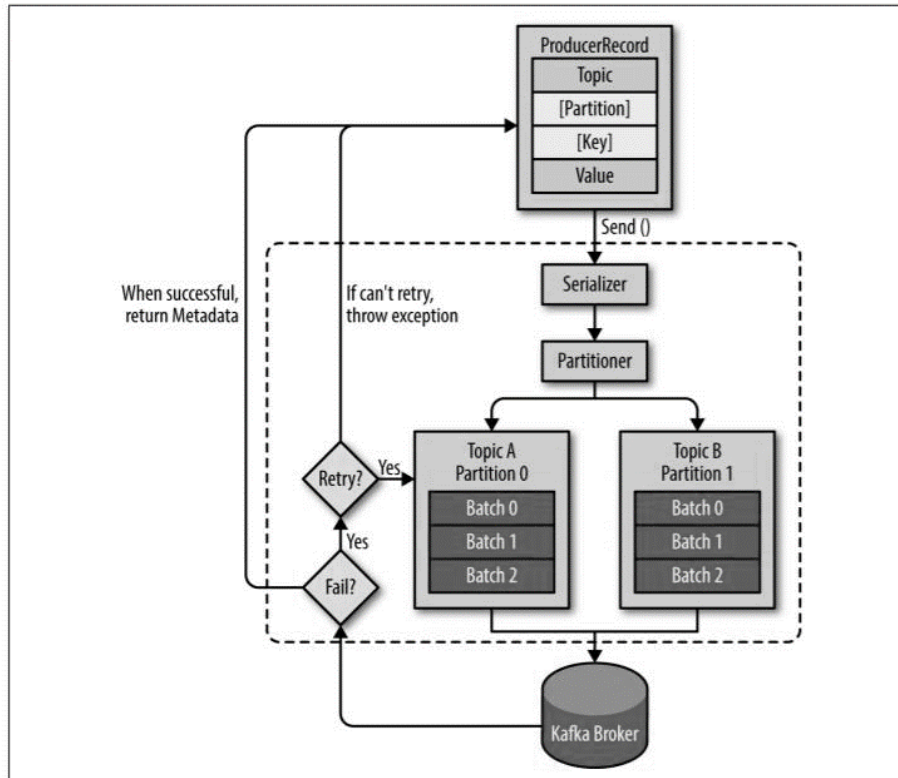
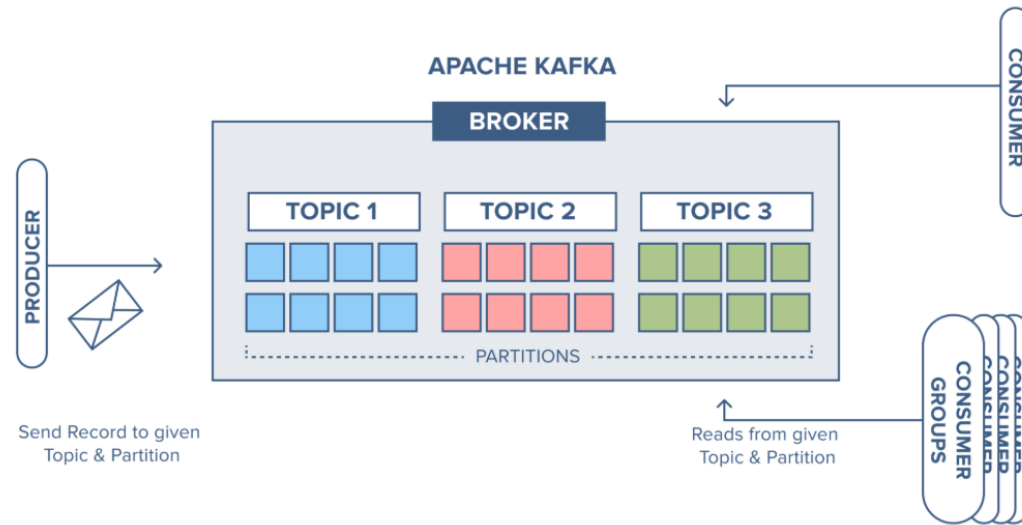


Figure 3-1. High-level overview of Kafka producer components



See <https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html> (last accessed March 15, 2024).

As an example, (on information and belief) see the following sample showing configured to identify a category of the update message based on the input source, available at <https://kafka.apache.org/23/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html> (last accessed March 15, 2024).

	<i>Method Detail</i> partition <pre>int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster)</pre> Compute the partition for the given record. Parameters: topic - The topic name org.apache.kafka.clients.producer Class ProducerRecord<K,V> java.lang.Object org.apache.kafka.clients.producer.ProducerRecord<K,V>
<i>Id.</i>	
<i>Id.</i>	<i>Constructor Summary</i> Constructors Constructor and Description ProducerRecord(String topic, Integer partition, K key, V value) Creates a record to be sent to a specified topic and partition ProducerRecord(String topic, Integer partition, K key, V value, Iterable<Header> headers) Creates a record to be sent to a specified topic and partition ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value) Creates a record with a specified timestamp to be sent to a specified topic and partition ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value, Iterable<Header> headers) Creates a record with a specified timestamp to be sent to a specified topic and partition ProducerRecord(String topic, K key, V value) Create a record to be sent to Kafka ProducerRecord(String topic, V value) Create a record with no key
<i>Id.</i>	

	Producers are those client applications that publish (write) events to Kafka, and consumers are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once. Events are organized and durably stored in topics. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be "payments". Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine. • The Producer API to publish (write) a stream of events to one or more Kafka topics. • The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them. See https://kafka.apache.org/intro (last accessed on November 10, 2023).
--	---

2.4 Connect API

The Connect API allows implementing connectors that continually pull from some source data system into Kafka or push from Kafka into some sink data system.

Many users of Connect won't need to use this API directly, though, they can use pre-built connectors without needing to write any code. Additional information on using Connect is available [here](#).

See <https://kafka.apache.org/documentation/> (last accessed on November 10, 2023).

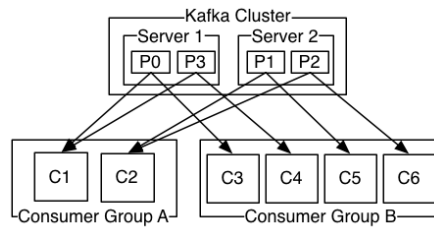
As an example (on information and belief), the following usage example provides a sample of the claim limitation. For all consumer instances reading from the same topic, they are grouped with distinctive IDs. An even distribution assignment of partitions of the topics will happen across each consumer group. The consumer API offers the functionality of offset committing to provide an accurate consumption of the messages in a topic stream.

Each consumer in a group can dynamically set the list of topics it wants to subscribe to through one of the `subscribe` APIs. Kafka will deliver each message in the subscribed topics to one process in each consumer group. This is achieved by balancing the partitions between all members in the consumer group so that each partition is assigned to exactly one consumer in the group. So if there is a topic with four partitions, and a consumer group with two processes, each process would consume from two partitions.

See <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html> (last accessed March 19, 2024).



	Usage Examples The consumer APIs offer flexibility to cover a variety of consumption use cases. Here are some examples to demonstrate how to use them. Automatic Offset Committing This example demonstrates a simple usage of Kafka's consumer api that relying on automatic offset committing. <pre>Properties props = new Properties(); props.put("bootstrap.servers", "localhost:9092"); props.put("group.id", "test"); props.put("enable.auto.commit", "true"); props.put("auto.commit.interval.ms", "1000"); props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer"); KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props); consumer.subscribe(Arrays.asList("foo", "bar")); while (true) { ConsumerRecords<String, String> records = consumer.poll(100); for (ConsumerRecord<String, String> record : records) System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value()); }</pre> The connection to the cluster is bootstrapped by specifying a list of one or more brokers to contact using the configuration <code>>bootstrap.servers</code>. This list is just used to discover the rest of the brokers in the cluster and need not be an exhaustive list of servers in the cluster (though you may want to specify more than one in case there are servers down when the client is connecting). Setting <code>enable.auto.commit</code> means that offsets are committed automatically with a frequency controlled by the config <code>auto.commit.interval.ms</code>. In this example the consumer is subscribing to the topics <i>foo</i> and <i>bar</i> as part of a group of consumers called <i>test</i> as configured with <code>group.id</code>. <i>Id.</i>; See also https://kafka.apache.org/20/documentation.html (last accessed March 15, 2024) (detailing Consumer API). • The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them. <i>Id.</i>
--	---



See <https://www.confluent.io/blog/a-comprehensive-rest-proxy-for-kafka/> (last accessed March 15, 2024).

Source Code Evidence

[REDACTED]

14[e] wherein the client device processes the update message upon receipt to update the property of the live object at the client device.	Upon information and belief, Comerica’s systems include “wherein the client device processes the update message upon receipt to update the property of the live object at the client device.” Upon receiving the events from one or more topics, the consumer entity/service/application processes the events (e.g. client device processes the update message). These consumed update messages are aggregated, enriched, or transformed into a new topic (e.g. update the property of the live object). <u>Kafka APIs</u> In addition to command line tooling for management and administration tasks, Kafka has five core APIs for Java and Scala: • The Admin API to manage and inspect topics, brokers, and other Kafka objects. • The Producer API to publish (write) a stream of events to one or more Kafka topics. • The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them. <u>Stream Processing</u> Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might normalize or deduplicate this content and publish the cleansed article content to a new topic; a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics. Starting in 0.10.0.0, a light-weight but powerful stream processing library called Kafka Streams is available in Apache Kafka to perform such data processing as described above. Apart from Kafka Streams, alternative open source stream processing tools include Apache Storm and Apache Samza.
--	---

See <https://kafka.apache.org/uses> (last accessed March 21, 2024).

As an example (on information and belief), the following usage example provides a sample of the claim limitation. For all consumer instances reading from the same topic, they are grouped with distinctive IDs. An even distribution assignment of partitions of the topics will happen across each consumer group. The consumer API offers the functionality of offset committing to provide an accurate consumption of the messages in a topic stream.

Each consumer in a group can dynamically set the list of topics it wants to subscribe to through one of the `subscribe` APIs. Kafka will deliver each message in the subscribed topics to one process in each consumer group. This is achieved by balancing the partitions between all members in the consumer group so that each partition is assigned to exactly one consumer in the group. So if there is a topic with four partitions, and a consumer group with two processes, each process would consume from two partitions.

Usage Examples

The consumer APIs offer flexibility to cover a variety of consumption use cases. Here are some examples to demonstrate how to use them.

Automatic Offset Committing

This example demonstrates a simple usage of Kafka's consumer api that relying on automatic offset committing.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
}
```

The connection to the cluster is bootstrapped by specifying a list of one or more brokers to contact using the configuration `>bootstrap.servers`. This list is just used to discover the rest of the brokers in the cluster and need not be an exhaustive list of servers in the cluster (though you may want to specify more than one in case there are servers down when the client is connecting).

Setting `enable.auto.commit` means that offsets are committed automatically with a frequency controlled by the config `auto.commit.interval.ms`.

In this example the consumer is subscribing to the topics `foo` and `bar` as part of a group of consumers called `test` as configured with `group.id`.

See <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html>.



Once we create a consumer, the next step is to subscribe to one or more topics. The `subscribe()` method takes a list of topics as a parameter, so it's pretty simple to use:

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

- ❶ Here we simply create a list with a single element: the topic name `customerCountries`.

It is also possible to call `subscribe` with a regular expression. The expression can match multiple topic names, and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic. This is useful for applications that need to consume from multiple topics and can handle the different types of data the topics will contain. Subscribing to multiple topics using a regular expression is most commonly used in applications that replicate data between Kafka and another system or streams processing applications.

See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

As another example, the below guidebook provides another example showing this limitation.

Chapter 4. Kafka Consumers: Reading Data from Kafka

Applications that need to read data from Kafka use a `KafkaConsumer` to subscribe to Kafka topics and receive messages from these topics. Reading data from Kafka is a bit different than reading data from other messaging systems, and there are few unique concepts and ideas involved. It is difficult to understand how to use the consumer API without understanding these concepts first. We'll start by explaining some of the important concepts, and then we'll go through some examples that show the different ways consumer APIs can be used to implement applications with varying requirements.

Commits and Offsets

Whenever we call `poll()`, it returns records written to Kafka that consumers in our group have not read yet. This means that we have a way of tracking which records were read by a consumer of the group. As discussed before, one of Kafka's unique characteristics is that it does not track acknowledgments from consumers the way many JMS queues do. Instead, it allows consumers to use Kafka to track their position (offset) in each partition.

We call the action of updating the current position in the partition a **commit**.

How does a consumer commit an offset? It produces a message to Kafka, to a special `__consumer_offsets` topic, with the committed offset for each partition. As long as all your consumers are up, running, and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will *trigger a rebalance*. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset of each partition and continue from there.

If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice. See Figure 4-6.

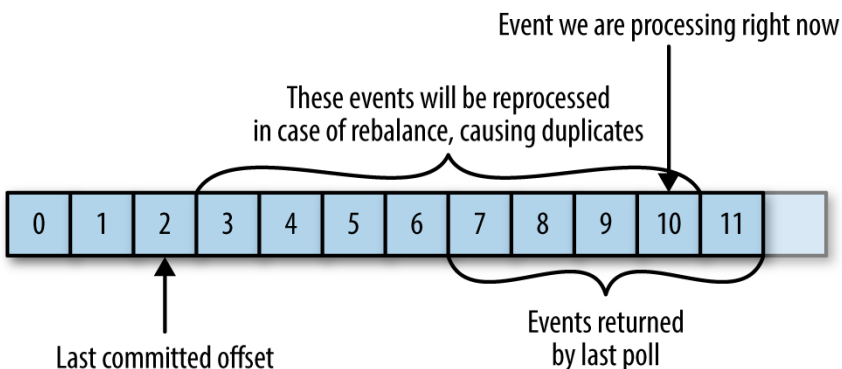


Figure 4-6. Re-processed messages

If the committed offset is larger than the offset of the last message the client actually processed, all messages between the last processed offset and the committed offset will be missed by the consumer group. See Figure 4-7.

See <https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html> (last accessed March 11, 2024).

Source Code Evidence

[REDACTED]

	[REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
Claim 16	Comerica Count II Systems and Services
16. The method of claim 14, wherein providing the data representation to the client device includes providing the live object that causes the client device to register with the node of the routing network.	Upon information and belief, Comerica’s systems perform the “method of claim 14, wherein providing the data representation to the client device includes providing the live object that causes the client device to register with the node of the routing network.” On information and belief, Kafka facilitates the performance of this claim by Kafka clients performing DNS resolution. <i>See Claim 14, supra.</i> Kafka has four core APIs: • The Producer API allows an application to publish a stream of records to one or more Kafka topics. • The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them. <u>See https://kafka.apache.org/20/documentation.html</u> (last accessed March 11, 2024).



Usage Examples

The consumer APIs offer flexibility to cover a variety of consumption use cases. Here are some examples to demonstrate how to use them.

Automatic Offset Committing

This example demonstrates a simple usage of Kafka's consumer api that relying on automatic offset committing.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());
}
```

The connection to the cluster is bootstrapped by specifying a list of one or more brokers to contact using the configuration `>bootstrap.servers`. This list is just used to discover the rest of the brokers in the cluster and need not be an exhaustive list of servers in the cluster (though you may want to specify more than one in case there are servers down when the client is connecting).

Setting `enable.auto.commit` means that offsets are committed automatically with a frequency controlled by the config `auto.commit.interval.ms`.

In this example the consumer is subscribing to the topics `foo` and `bar` as part of a group of consumers called `test` as configured with `group.id`.

See <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html> (last accessed March 11, 2024).

As an example (on information and belief), Kafka validates, resolves, and creates connections based on the hostname. Kafka clients perform DNS resolution to help establish the connection to a broker. To enable an easier connection configuration, a single DNS alias to address all brokers in a cluster may be created.

By default, Kafka validates, resolves, and creates connections based on the hostname provided in the bootstrap server configuration (and later in the names returned by the brokers as specified in the `advertised.listeners` configuration). This simple model works most of the time but fails to cover two important use cases: the use of DNS aliases, especially in a bootstrap configuration, and the use of a single DNS that maps to multiple IP addresses. These sound similar but are slightly different. Let's look at each of these mutually exclusive scenarios in a bit more detail.

Suppose you have multiple brokers with the following naming convention: `broker1.hostname.com`, `broker2.hostname.com`, etc. Rather than specifying all of them in a bootstrap server configuration, which can easily become challenging to maintain, you may want to create a single DNS alias that will map to all of them. You'll use `all-brokers.hostname.com` for bootstrapping, since you don't actually care which broker gets the initial connection from clients. This is all very convenient, except if you use SASL to authenticate. If you use SASL, the client will try to authenticate `all-brokers.hostname.com`, but the server principal will be `broker2.hostname.com`. If the names don't match, SASL will refuse to authenticate (the broker certificate could be a man-in-the-middle attack), and the connection will fail.

In this scenario, you'll want to use `client.dns.lookup=resolve_canonical_bootstrap_servers_only`. With this configuration, the client will "expend" the DNS alias, and the result will be the same as if you included all the broker names the DNS alias connects to as brokers in the original bootstrap list.

See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

As a further example, (on information and belief) Kafka clients perform DNS resolution to help establish the connection to a broker.

DNS Resolution for Kafka clients

DNS resolution is an important aspect of Kafka client connectivity as it enables the clients to connect to the correct Kafka brokers. When a Kafka client is configured to connect to a Kafka cluster, it typically specifies a list of broker hostnames or IP addresses in its configuration.

When a Kafka client is started, it first reads its configuration file to determine the initial list of bootstrap brokers. These brokers are specified as a comma-separated list of hostname and port pairs (e.g.,

```
broker1.example.com:9092,broker2.example.com:9092 ).
```

The Kafka client then performs DNS resolution to obtain the IP addresses of the specified hostnames. By default, the client uses the operating system's default DNS resolution behavior, which typically involves sending DNS queries to a DNS resolver provided by the operating system.

See <https://levelup.gitconnected.com/dns-resolution-for-kafka-clients-2e76b28e4717> (last accessed March 11, 2024).

Source Code Evidence

[REDACTED]

[REDACTED]

	[REDACTED]
Claim 17	Comerica Count II Systems and Services
17. The method of claim 14, wherein providing the data representation to the client device includes providing an activation module that is executed by the client device and that registers the live object with the routing network.	Upon information and belief, Comerica’s systems perform the “method of claim 14, wherein providing the data representation to the client device includes providing an activation module that is executed by the client device and that registers the live object with the routing network.” On information and belief, Kafka facilitates the performance of this claim. <i>See Claim 14, supra.</i> As an example, the Consumer API allows an application to process the stream of records produced to them. https://kafka.apache.org/20/documentation.html (last accessed March 11, 2024). Kafka has four core APIs: • The Producer API allows an application to publish a stream of records to one or more Kafka topics. • The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them. <i>Id.</i> As a further example (on information and belief) the below usage example shows how this limitation is met:



Usage Examples

The consumer APIs offer flexibility to cover a variety of consumption use cases. Here are some examples to demonstrate how to use them.

Automatic Offset Committing

This example demonstrates a simple usage of Kafka's consumer api that relying on automatic offset committing.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
}
```

See

<https://kafka.apache.org/24/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html#:~:text=Consumer%20Groups%20and%20Topic%20Subscriptions&text=Each%20consumer%20in%20a%20group,process%20in%20each%20consumer%20group> (last accessed March 11, 2024).

The connection to the cluster is bootstrapped by specifying a list of one or more brokers to contact using the configuration `>bootstrap.servers`. This list is just used to discover the rest of the brokers in the cluster and need not be an exhaustive list of servers in the cluster (though you may want to specify more than one in case there are servers down when the client is connecting).

Setting `enable.auto.commit` means that offsets are committed automatically with a frequency controlled by the config `auto.commit.interval.ms`.

In this example the consumer is subscribing to the topics *foo* and *bar* as part of a group of consumers called *test* as configured with `group.id`.

Id.

See also <https://www.conduktor.io/kafka/complete-kafka-consumer-with-java/> (last accessed March 15, 2024) (describing an example using the feature that subscribes the consumer to a specific topic):

Subscribe the consumer to a specific topic

To read the messages from a topic, we need to connect the consumer to the specified topic. Here, we use `Arrays.asList()`, as it allows our consumer to subscribe to multiple topics.

Below code shows the implementation of subscription of the consumer to one topic:

```
1 // subscribe consumer to our topic(s)
2 consumer.subscribe(Arrays.asList(topic));
```

Id.

As a further example (on information and belief), Kafka destination entities/services/applications (e.g. client device) initializing a consumer instance subscribing to topics are also noted as subscribers.

Once we create a consumer, the next step is to subscribe to one or more topics. The `subscribe()` method takes a list of topics as a parameter, so it's pretty simple to use:

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

- ❶ Here we simply create a list with a single element: the topic name `customerCountries`.

It is also possible to call `subscribe` with a regular expression. The expression can match multiple topic names, and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic. This is useful for applications that need to consume from multiple topics and can handle the different types of data the topics will contain. Subscribing to multiple topics using a regular expression is most commonly used in applications that replicate data between Kafka and another system or streams processing applications.

For example, Kafka validates, resolves, and creates connections based on the hostname. Kafka clients perform DNS resolution to help establish the connection to a broker. To enable an easier connection configuration, a single DNS alias to address all brokers in a cluster may be created.

By default, Kafka validates, resolves, and creates connections based on the hostname provided in the bootstrap server configuration (and later in the names returned by the brokers as specified in the `advertised.listeners` configuration). This simple model works most of the time but fails to cover two important use cases: the use of DNS aliases, especially in a bootstrap configuration, and the use of a single DNS that maps to multiple IP addresses. These sound similar but are slightly different. Let's look at each of these mutually exclusive scenarios in a bit more detail.

Suppose you have multiple brokers with the following naming convention: broker1.hostname.com, broker2.hostname.com, etc. Rather than specifying all of them in a bootstrap server configuration, which can easily become challenging to maintain, you may want to create a single DNS alias that will map to all of them.

You'll use all-brokers.hostname.com for bootstrapping, since you don't actually care which broker gets the initial connection from clients. This is all very convenient, except if you use SASL to authenticate. If you use SASL, the client will try to authenticate all-brokers.hostname.com, but the server principal will be broker2.hostname.com. If the names don't match, SASL will refuse to authenticate (the broker certificate could be a man-in-the-middle attack), and the connection will fail.

In this scenario, you'll want to use client.dns.lookup=resolve_canonical_bootstrap_servers_only. With this configuration, the client will "expend" the DNS alias, and the result will be the same as if you included all the broker names the DNS alias connects to as brokers in the original bootstrap list.

See Neha Narkhede, Gwen Shapira, Todd Palino. 2017. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc.

Source Code Evidence

[REDACTED]

	[REDACTED]
Claim 18	Comerica Count II Systems and Services
18. The method of claim 17, wherein providing the activation module includes providing the activation module that is configured to determine a node type that handles registration and that causes the client device to register with the node corresponding to the node type.	Upon information and belief, Comerica’s systems perform the “method of claim 17, wherein providing the activation module includes providing the activation module that is configured to determine a node type that handles registration and that causes the client device to register with the node corresponding to the node type.” On information and belief, Kafka facilitates the performance of this claim. <i>See Claim 17, supra.</i> As an example (on information and belief), Kafka provides offsets including automatic offsets, which determine a node type that handles registration and that causes the client device to register with the node corresponding to the node type. We start producing messages to Kafka by creating a <code>ProducerRecord</code>, which must include the topic we want to send the record to and a value. Optionally, we can also specify a key and/or a partition. Once we send the <code>ProducerRecord</code>, the first thing the producer will do is serialize the key and value objects to <code>ByteArrays</code> so they can be sent over the network. Next, the data is sent to a partitioner. If we specified a partition in the <code>ProducerRecord</code>, the partitioner doesn’t do anything and simply returns the partition we specified. If we didn’t, the partitioner will choose a partition for us, usually based on the <code>ProducerRecord</code> key. Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition. A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers. <i>See https://www.confluent.io/resources/kafka-the-definitive-guide/ (last accessed March 11, 2024)</i>

Usage Examples

The consumer APIs offer flexibility to cover a variety of consumption use cases. Here are some examples to demonstrate how to use them.

Automatic Offset Committing

This example demonstrates a simple usage of Kafka's consumer api that relying on automatic offset committing.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
}
```

See

<https://kafka.apache.org/24/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html>

(last accessed March 11, 2024)

The connection to the cluster is bootstrapped by specifying a list of one or more brokers to contact using the configuration `>bootstrap.servers`. This list is just used to discover the rest of the brokers in the cluster and need not be an exhaustive list of servers in the cluster (though you may want to specify more than one in case there are servers down when the client is connecting).

Setting `enable.auto.commit` means that offsets are committed automatically with a frequency controlled by the config `auto.commit.interval.ms`.

In this example the consumer is subscribing to the topics *foo* and *bar* as part of a group of consumers called *test* as configured with `group.id`.

Id.

As a further example (on information and belief), Kafka destination entities/services/applications (e.g. client device) initializing a consumer instance subscribing to topics are also noted as subscribers. The

KafkaConsumer class with the consumer API maintains an offset committing to provide an accurate consumption of the messages in a topic stream.

```
org.apache.kafka.clients.consumer
Class KafkaConsumer<K,V>
java.lang.Object
org.apache.kafka.clients.consumer.KafkaConsumer<K,V>
```

See <https://kafka.apache.org/22/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html> (last accessed March 11, 2024)

Offsets and Consumer Position

Kafka maintains a numerical offset for each record in a partition. This offset acts as a unique identifier of a record within that partition, and also denotes the position of the consumer in the partition. For example, a consumer which is at position 5 has consumed records with offsets 0 through 4 and will next receive the record with offset 5. There are actually two notions of position relevant to the user of the consumer:

Id.

Source Code Evidence

[Redacted Source Code Evidence]



--	--