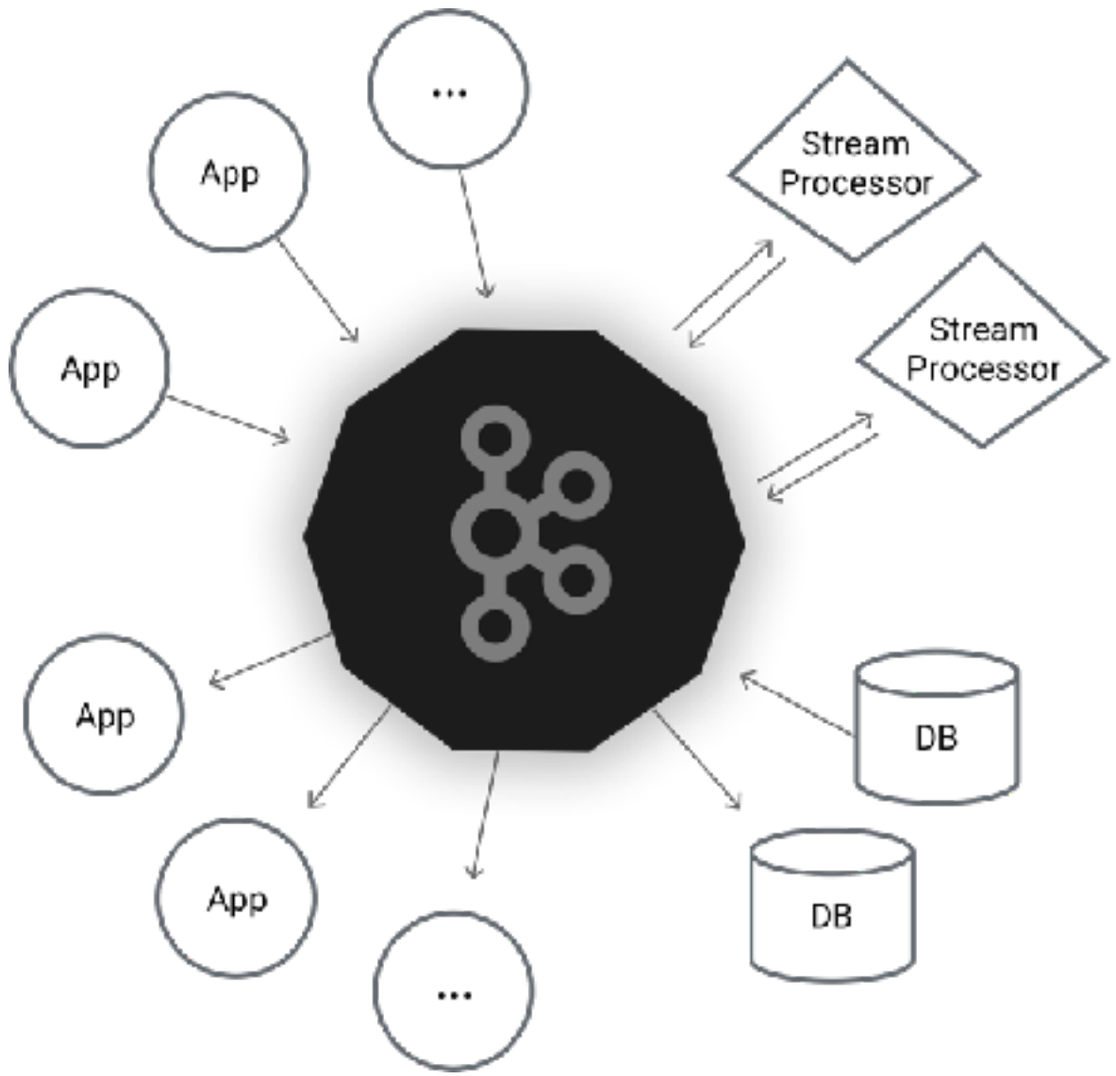
APACHE
kafka®
A distributed streaming platform

Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.



APACHE kafka®
A distributed streaming platform

Messaging System

A messaging system is one of the most commonly used mechanisms for information exchange in applications

(The other mechanisms used to share information could be remote procedure calls (RPC), file share, shared databases, and web service invocation)

Enterprises have started adopting micro service architecture and the main advantage of doing so is to make applications loosely coupled with each other

APACHE kafka
A distributed streaming platform

Definitions

Messages (data packets): A message is an atomic data packet that gets transmitted over a network to a message queue

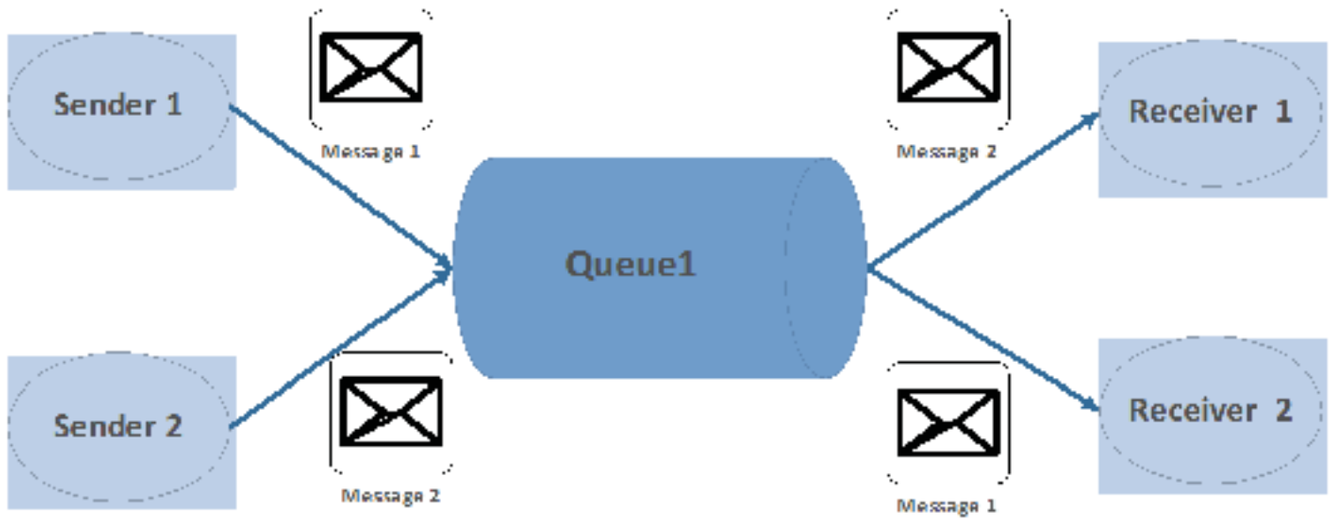
Sender (producer): Sender or producer applications are the sources of data that needs to be sent to a certain destination

Receiver (consumer): Receiver or consumer applications are the receivers of messages sent by the sender application.

Data transmission protocols: Data transmission protocols determine rules to govern message exchanges between applications (Kafka uses binary protocols over TCP)

Peeking into a point-to-point messaging system

Point-to-point messaging is generally used when a single message will be received by only one message consumer .You can think of queues supporting PTP messaging models as FIFO queues .Queues such as Kafka maintain message offsets. Instead of deleting the messages, they increment the offsets for the receiver. Offset-based models provide better support for replaying messages



Peeking into a point-to-point messaging system

The PTP messaging model can be further categorized into two types:

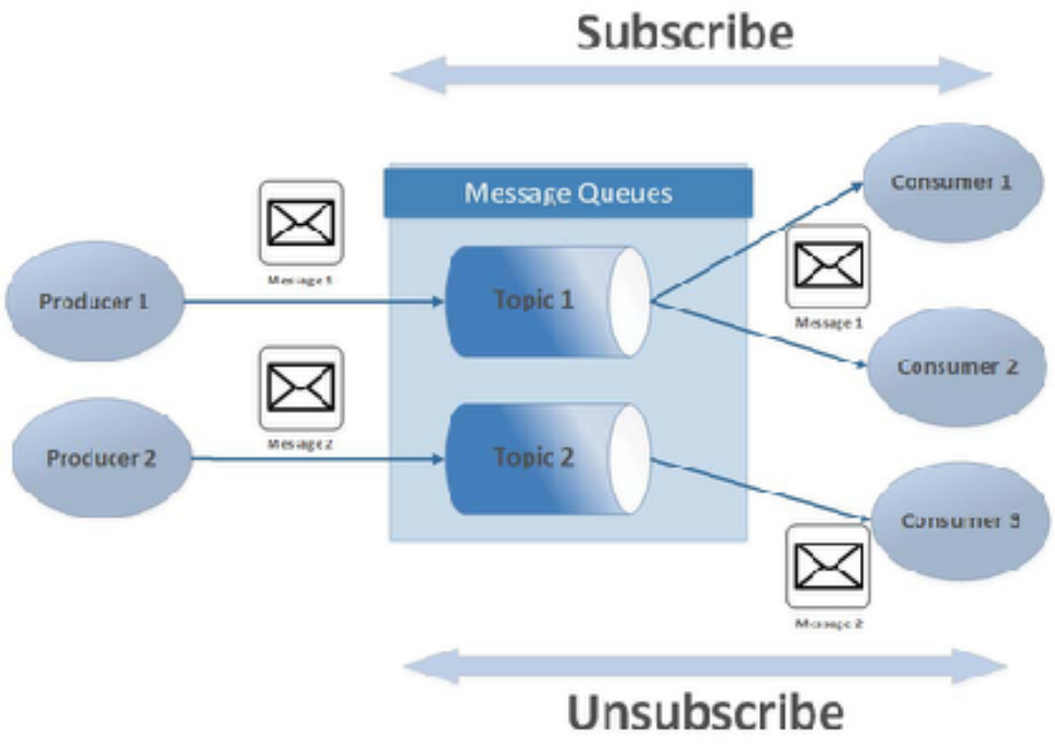
Fire-and-forget model

Request/reply model

The request/reply model provides for a high degree of decoupling between the sender and receiver, allowing the message producer and consumer components to be heterogeneous languages or platforms

Publish-subscribe messaging system

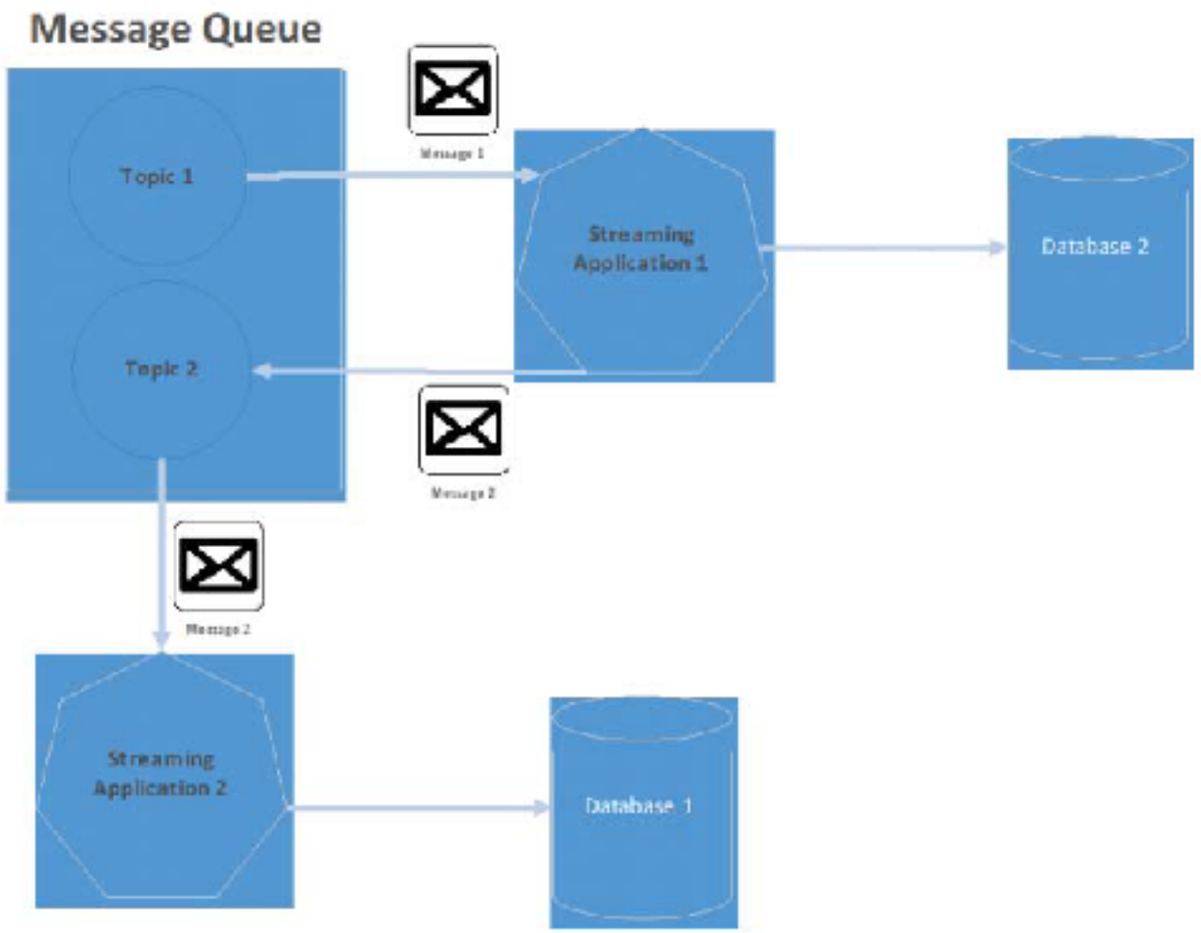
A subscriber registers its interest in a particular topic or event and is subsequently notified about the event asynchronously.It is different from the PTP messaging model in a way that a topic can have multiple receivers and every receiv receives a copy of each message



*Messages are shared through a channel called a **topic**. A topic is a centralized place where producers can publish, and subscribers can consume, messages. Each message is delivered to one or more message consumers, called **subscribers**.There is no coupling of the producers to the consumers. Subscribers and publishers can be added dynamically at runtime, which allows the system to grow or shrink in complexity over time.*

APACHE kafka
A distributed streaming platform

Using messaging systems in big data streaming applications



We need a messaging system that immediately tells the streaming application that,
Something got published; please process it.

These are the points that are important for any streaming application

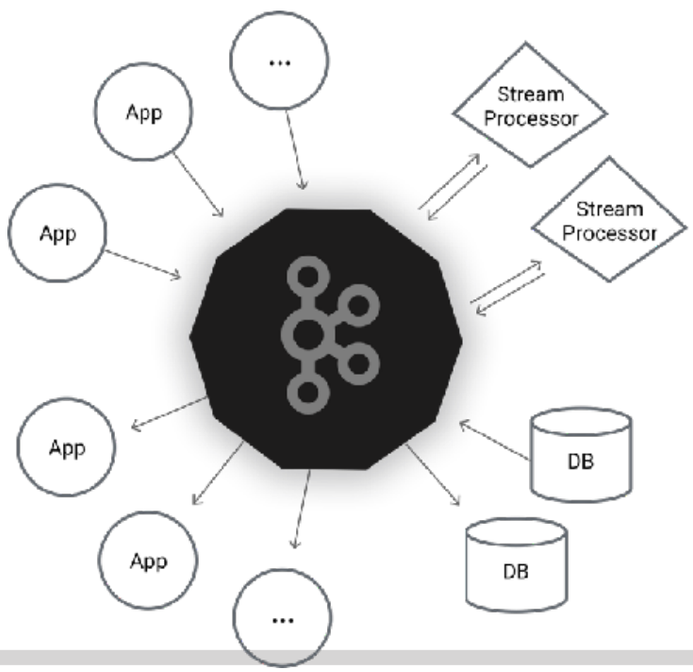
High consuming rate
Guaranteed delivery
Persisting capability:
Security:
Fault tolerance

APACHE kafka
A distributed streaming platform

Introducing Kafka

Kafka was released as an open source project on GitHub in late 2010. As it started to gain attention in the open source community, it was proposed and accepted as an Apache Software Foundation incubator project in July of 2011

Kafka was built with the following **goals** in mind:
Loose coupling between message Producers and message Consumers
Persistence of message data to support a variety of data consumption scenarios and failure handling
Maximum end-to-end throughput with low latency components
Managing diverse data formats and types using binary data formats
Scaling servers linearly without affecting the existing cluster setup



APACHE kafka®
A distributed streaming platform

Use Case

Activity tracking

The original use case for Kafka, as it was designed at *LinkedIn*, is that of user activity tracking. A website's users interact with frontend applications, which generate mes- sages regarding actions the user is taking.These applications may be generating reports, feeding machine learning systems, updating search results, or per- forming other operations that are necessary to provide a rich user experience.

Messaging

Kafka is also used for messaging, where applications need to send notifications (such as emails) to users. Those applications can produce messages without needing to be concerned about formatting or how the messages will actually be sent

Commit log

Since Kafka is based on the concept of a commit log, database changes can be published to Kafka and applications can easily monitor this stream to receive live updates as they happen. This changelog stream can also be used for replicating database updates to a remote system, or for consolidating changes from multiple applications into a single database view.

APACHE kafka
A distributed streaming platform

Use Case

Metrics and logging

Kafka is also ideal for collecting application and system metrics and logs. This is a use case in which the ability to have multiple applications producing the same type of message Applications publish metrics on a regular basis to a Kafka topic, and those metrics can be consumed by systems for monitoring and alerting. They can also be used in an offline system like Hadoop to perform longer-term analysis, such as growth projections. Log messages can be published in the same way, and can be routed to dedicated log search systems like Elastisearch or security analysis applications.

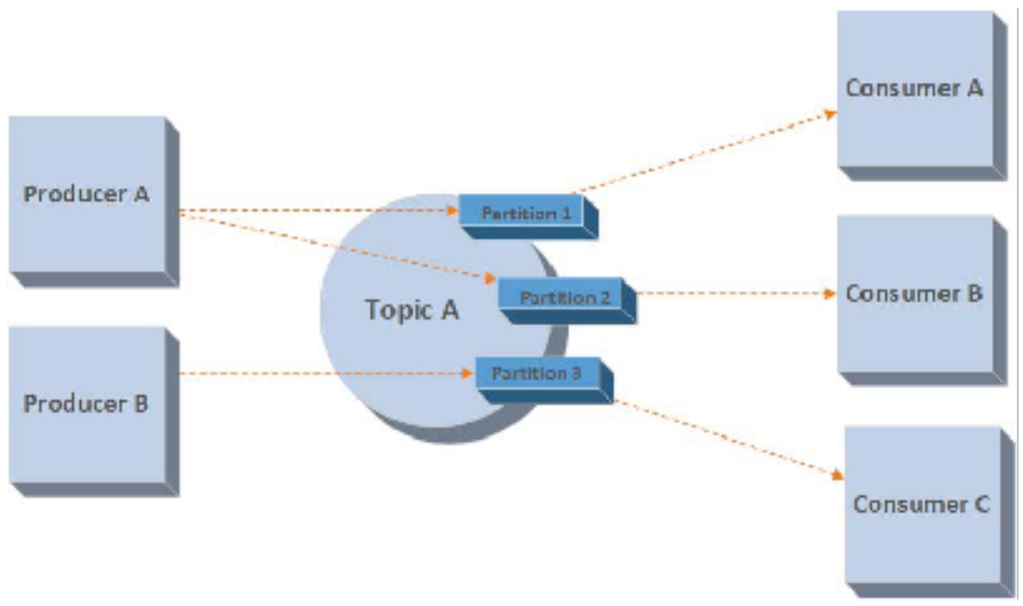
Stream processing

While almost all usage of Kafka can be thought of as stream processing, the term is typically used to refer to applications that provide similar functionality to map/reduce processing in Hadoop. *Hadoop* usually relies on aggregation of data over a long time frame, either hours or days. Stream processing operates on data in real time, as quickly as messages are produced. Stream frameworks allow users to write small applications to operate on Kafka messages, performing tasks such as counting metrics, partitioning messages for efficient processing by other applications, or transforming messages using data from multiple sources.

Kafka's architecture

Every message in Kafka topics is a collection of bytes. This collection is represented as an array. Producers are the applications that store information in **Kafka queues**. They send messages to Kafka topics that can store all types of messages. Every topic is further differentiated into **partitions**. Each partition stores messages in the sequence in which they arrive.

There are two major operations that producers/consumers can perform in Kafka.
Producers append to the end of the write-ahead log files.
Consumers fetch messages from these log files belonging to a given topic partition.
Physically, each topic is spread over
different Kafka brokers, which host one or two partitions of each topic.



APACHE kafka
A distributed streaming platform

Kafka's architecture

A Kafka cluster is basically composed of one or more servers (nodes) also knows as broker.

The **broker** receives messages from producers, assigns offsets to them, and commits the messages to storage on disk.
Kafka persists all messages to disk, and these log segments are stored in the directories specified in the ***log.dirs*** configuration

It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk.

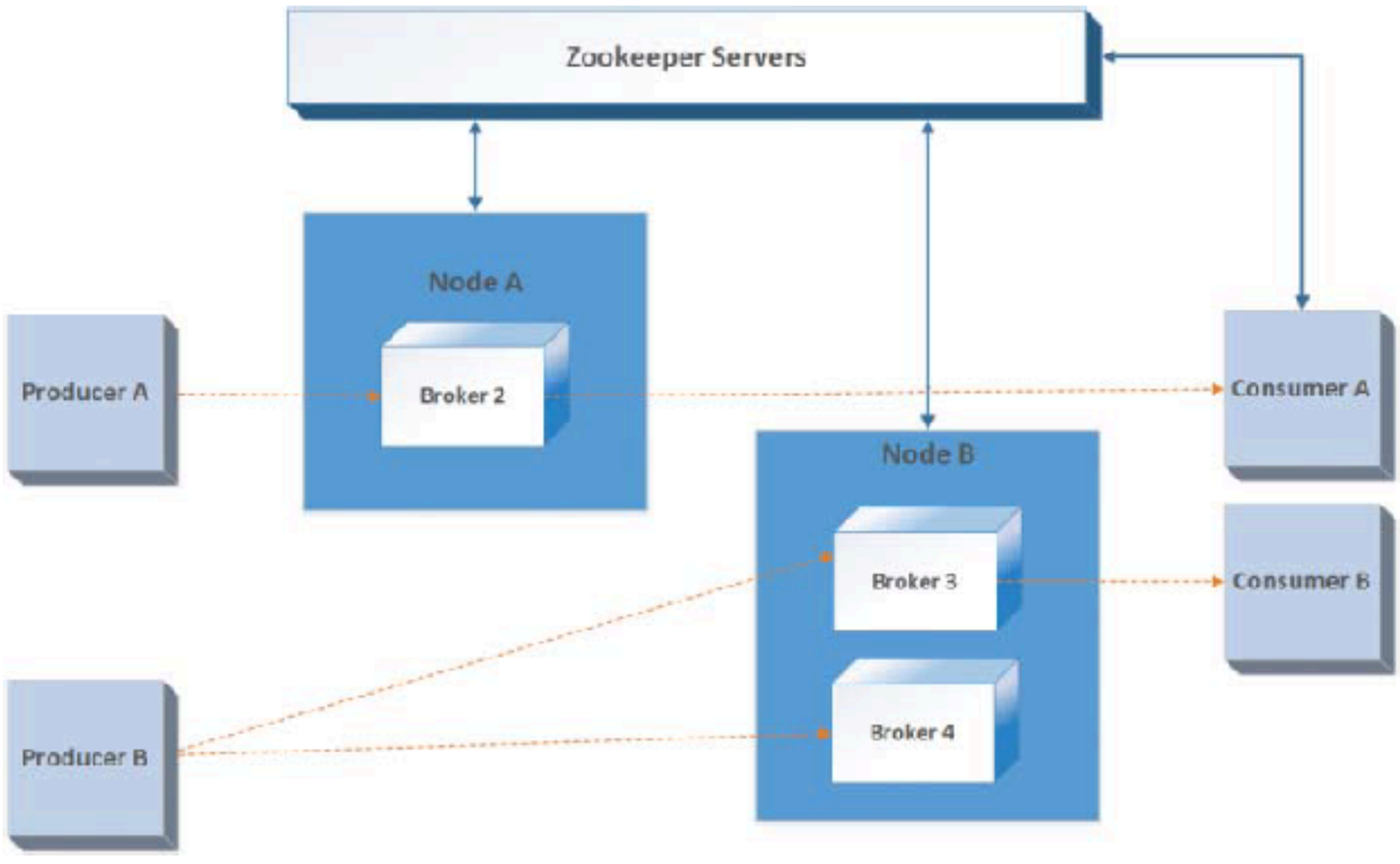
Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second

Kafka's architecture

A typical Kafka cluster consists of multiple brokers. It helps in load-balancing message reads and writes to the cluster.

Each of these brokers is **stateless**. However, they use **Zookeeper** to maintain their states.

Each topic partition has one of the brokers as a **leader** and zero or more brokers as followers.The leaders manage any read or write requests for their respective partitions



APACHE kafka®
A distributed streaming platform

Apache Zookeeper

Zookeeper is an important component of a Kafka cluster. It manages and coordinates Kafka brokers and consumers.
Zookeeper keeps track of any new broker additions or any existing broker failures in the Kafka cluster.
it will notify the producer or consumers of Kafka queues about the cluster state.
This helps both producers and consumers in coordinating work with active brokers.
Zookeeper also records which broker is the leader for which topic partition and passes on this information to the producer or consumer to read and write the messages.

The consumer records its state with the help of Zookeeper as Kafka brokers are stateless.

APACHE kafka
A distributed streaming platform

Apache Zookeeper

Kafka cannot work without Zookeeper. Kafka uses Zookeeper for the following functions:

Choosing a controller: The controller is one of the brokers responsible for partition management with respect to leader election, topic creation, partition creation, and replica management. When a node or server shuts down, Kafka controllers elect partition leaders from followers. Kafka uses Zookeeper's metadata information to elect a controller. Zookeeper ensures that a new controller is elected in case the current controller crashes.

Brokers metadata: Zookeeper records the state of each of the brokers that are part of the Kafka cluster. It records all relevant metadata about each broker in a cluster. The producer/consumer interacts with Zookeeper to get the broker's state.

Apache Zookeeper

Topic metadata: Zookeeper also records topic metadata such as the number of partitions, specific configuration parameters, and so on.

Client quota information: With newer versions of Kafka, quota features have been introduced. Quotas enforce byte-rate thresholds on clients to read and write messages to a Kafka topic. All the information and states are maintained by Zookeeper.

Kafka topic ACLs: Kafka has an in-built authorization module that is defined as Access Control Lists (ACLs). These ACLs determine user roles and what kind of read and write permissions each of these roles has on respective topics. Kafka uses Zookeeper to store all ACLs

Apache Zookeeper

A Zookeeper cluster is called an *ensemble*.

Due to the algorithm used, it is recommended that ensembles contain an odd number of servers (e.g., 3, 5, etc.) as a majority of ensemble members (a quorum) must be working in order for Zookeeper to respond to requests

This means that in a three-node ensemble, you can run with one node missing. With a five-node ensemble, you can run with two nodes missing

Message topics

Here are a few terms that we need to know:

Retention Period: The messages in the topic need to be stored for a defined period of time to save space irrespective of throughput. We can configure the retention period, which is by default seven days, to whatever number of days we choose. Kafka keeps messages up to the defined period of time and then ultimately deletes them.

Space Retention Policy: We can also configure Kafka topics to clear messages when the size reaches the threshold mentioned in the configuration

Offset: Each message in Kafka is assigned with a number called as an offset. Topics consist of many partitions. Each partition stores messages in the sequence in which they arrive. Consumers acknowledge messages with an offset, which means that all the messages before that message offset are received by the consumer.

Message topics

Partition: Each Kafka topic consists of a fixed number of partitions. During topic creation in Kafka, you need to configure the number of partitions. Partitions are distributed and help in achieving high throughput.

Compaction: Topic compaction was introduced in Kafka 0.8. There is no way to change previous messages in Kafka; messages only get deleted when the retention period is over.

Leader: Partitions are replicated across the Kafka cluster based on the replication factor specified. Each partition has a leader broker and followers and all the read write requests to the partition will go through the leader only. If the leader fails, another leader will get elected and the process will resume.

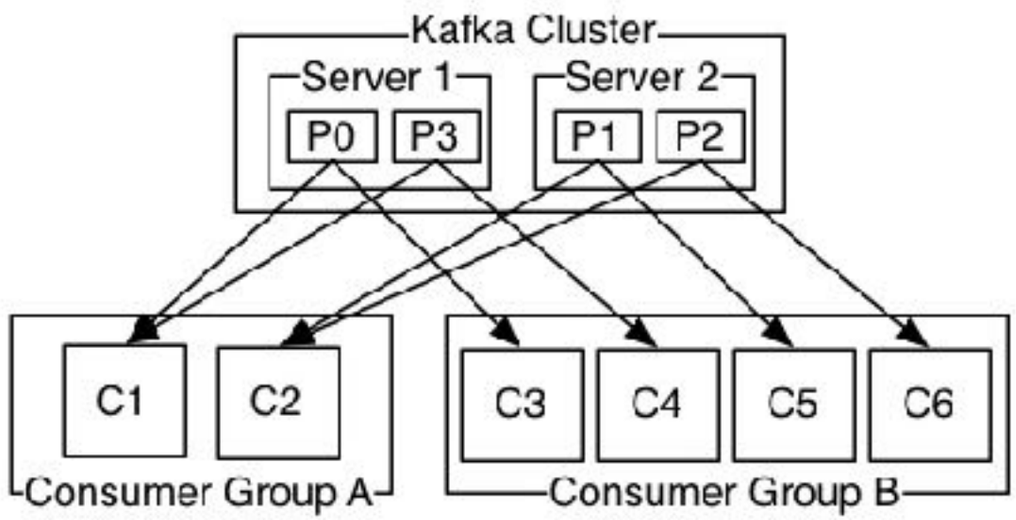
Buffering: Kafka buffers messages both at the producer and consumer side to increase throughput and reduce Input/Output (IO).

Message partitions

The topics are broken into partitions known as **units of parallelism** in Kafka.

This means that the greater the number of partitions, the more **throughput**. Each of the messages will be appended to partitions and each message is then assigned with a number called an offset.

Kafka makes sure that messages with similar keys always go to the same partition;
it calculates the hash of the message key and appends the message to the partition (Partitions are fault-tolerant; they are replicated across the Kafka brokers)



APACHE kafka
A distributed streaming platform

Message partitions

Pros and cons of a large number of partitions:

High throughput: Partitions are a way to achieve parallelism in Kafka. Write operations on different partitions happen in parallel. All time-consuming operations will happen in parallel as well; this operation will utilize hardware resources at the maximum

Increases producer memory: increasing the number of partitions will force us to increase producer memory. If we increase the number of partitions, the memory allocated for the buffering may exceed in a very short interval of time, and the producer will block producing messages until it sends buffered data to the broker

High availability issue:Kafka is known as high-availability, high-throughput, and distributed messaging system. Brokers in Kafka store thousands of partitions of different topics. Reading and writing to partitions happens through the leader of that partition. Generally, if the leader fails, electing a new leader takes only a few milliseconds. Observation of failure is done through controllers.
In the case of unexpected failure, such as killing a broker unintentionally, it may result in a delay of a few seconds based on the number of partitions.

The general formula is:
Delay Time = (Number of Partition/replication * Time to read metadata for single partition)

Replication and replicated logs

Replicas of message logs for each topic partition are maintained across different servers in a Kafka cluster.
This can be configured for each topic separately.
All the reads and writes happen through the leader; if the leader fails, one of the followers will be elected as leader.

To maintain **replica consistency**, there are two approaches

Quorum-based approach: In this approach, the leader will mark messages committed only when the majority of replicas have an acknowledged receiving the message. Zookeeper follows a quorum-based approach for leader election

Primary backup approach: Kafka follows a different approach to maintaining replicas; the leader in Kafka waits for an acknowledgement from all the followers before marking the message as committed. If the leader fails, any of the followers can take over as leader.
This approach can cost you more in terms of latency and throughput but this will guarantee better consistency for messages or data.

Kafka Producers

Kafka can be used as a *message queue*, *message bus*, *or data storage system*

The responsibilities of Kafka producers apart from publishing messages:

Bootstrapping Kafka broker URLs: The Producer connects to at least one broker to fetch metadata about the Kafka cluster.To ensure a failover, the producer implementation takes a list of more than one broker URL to bootstrap from.

Data serialization: Kafka uses a binary protocol to send and receive data over TCP. This means that while writing data to Kafka, producers need to send the ordered byte sequence to the defined Kafka broker's network port.
Kafka producer serializes every message data object into ByteArrays before sending any record to the respective broker.

Determining topic partition (with Zookeeper) : It is the responsibility of the Kafka producer to determine which topic partition data needs to be sent. If the partition is specified by the caller program, then Producer APIs do not determine topic partition and send data directly to it. However, if no partition is specified, then producer will choose a partition for the message.
This is generally based on the key of the message data object.

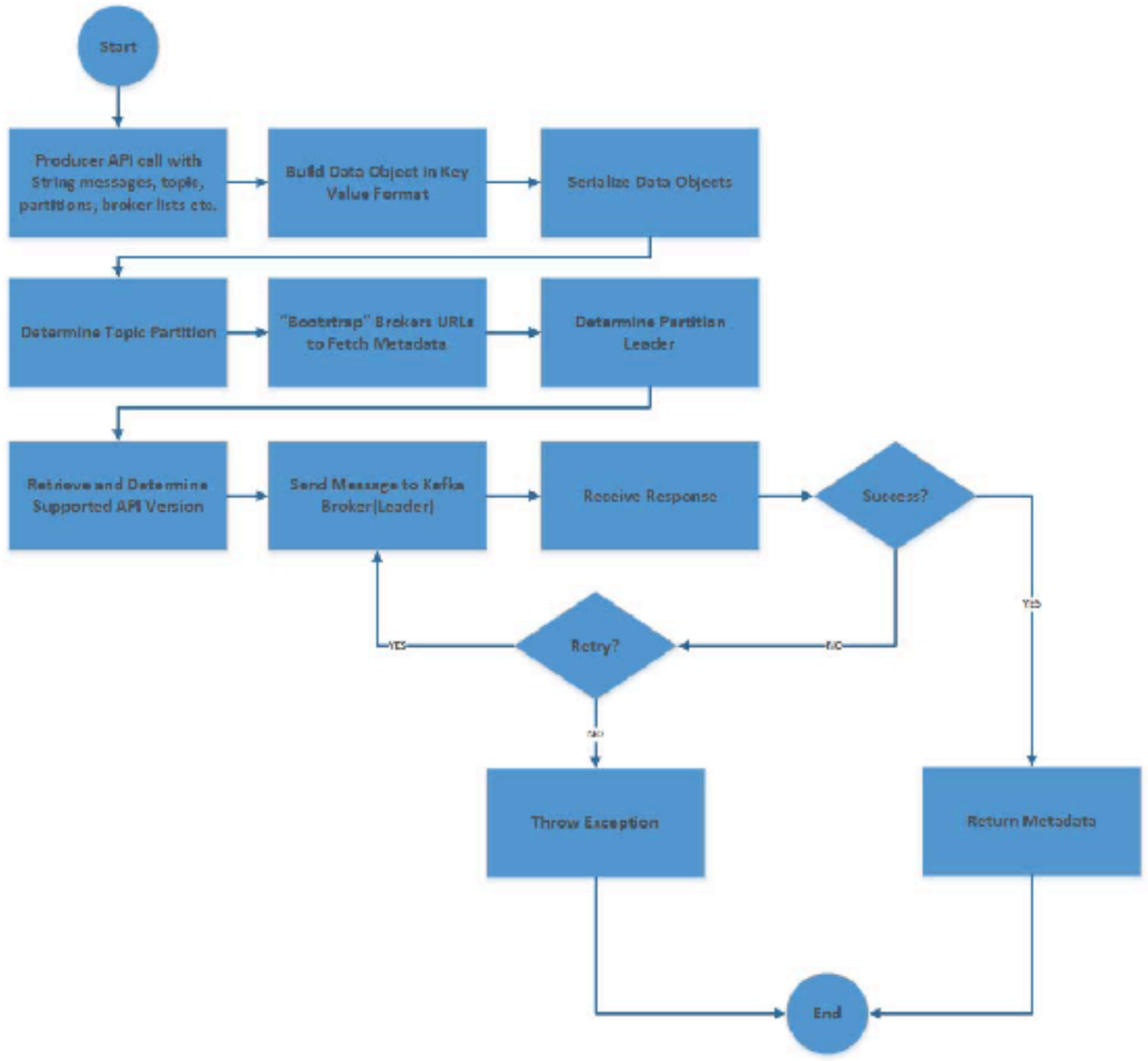
Kafka Producers

Determining the leader of the partition: Producers send data to the leader of the partition directly. It is the producer's responsibility to determine the leader of the partition to which it will write messages.

Failure handling/retry ability: Handling failure responses or number of retries is something that needs to be controlled through the producer application. You can configure the number of retries through Producer API configuration

Batching: For efficient message transfers, batching is a very useful mechanism. Through Producer API configurations, you can control whether you need to use the producer in asynchronous mode or not. Batching ensures reduced I/O and optimum utilization of producer memory

Kafka Producers



Batching/One At a Time

Kafka Producers API

Creating a Kafka producer involves the following steps:

1. Required configuration.
2. Creating a producer object.
3. Setting up a producer record.
4. Creating a custom partition if required.
5. Additional configuration.

The following are three mandatory configuration parameters:

bootstrap.servers:This contains a list of Kafka brokers addresses. The address is specified in terms of hostname:port. We can specify one or more broker detail, but we recommend that you provide at least two so that if one broker goes down, producer can use the other one

key.serializer: The message is sent to Kafka brokers in the form of a keyvalue pair. Brokers expect this key-value to be in byte arrays. So we need to tell producer which serializer class is to be used to convert this key-value object to a byte array
Kafka provides us with three inbuilt serializer classes: ByteArraySerializer, IntegerSerializer, StringSerializer

value.serializer this property tells the producer which class to use in order to serialize the value. You can implement your own serialize class and assign to this property

Kafka Producers API

Producer accepts the **ProducerRecord** object to send records to the ProducerRecord topic.

It contains a ***topic name, partition number, timestamp, key, and value***

If the partition number is specified, then the specified partition will be used when sending the record

If the partition is not specified but a key is specified, a partition will be chosen using a hash of the key

If both key and partition are not specified, a partition will be assigned in a **roundrobin fashion**.

If we do not mention a timestamp, the producer will stamp the record with its current time

Additional producer configuration

There are other optional configuration properties available for Kafka producer that can play an important role in *performance, memory, reliability*

buffer.memory:
This is the amount of memory that producer can use to buffer a message that is waiting to be sent to the Kafka server

acks
This configuration helps in configuring when producer will receive acknowledgment from the leader before considering that the message is committed successfully

batch.size
This setting allows the producer to batch the messages based on the partition up to the configured amount of size. When the batch reaches the limit, all messages in the batch will be sent

Additional producer configuration

linger.ms
Amount of time that a producer should wait for additional messages before sending a current batch to the broker. Kafka producer waits for the batch to be full or the configured linger.ms time

compression.time
By default, producer sends uncompressed messages to brokers. When sending a single message, it will not make that much sense, but when we use batches, it's good to use compression to avoid network overhead and increase throughput. The available compressions are **GZIP, Snappy, or LZ4**

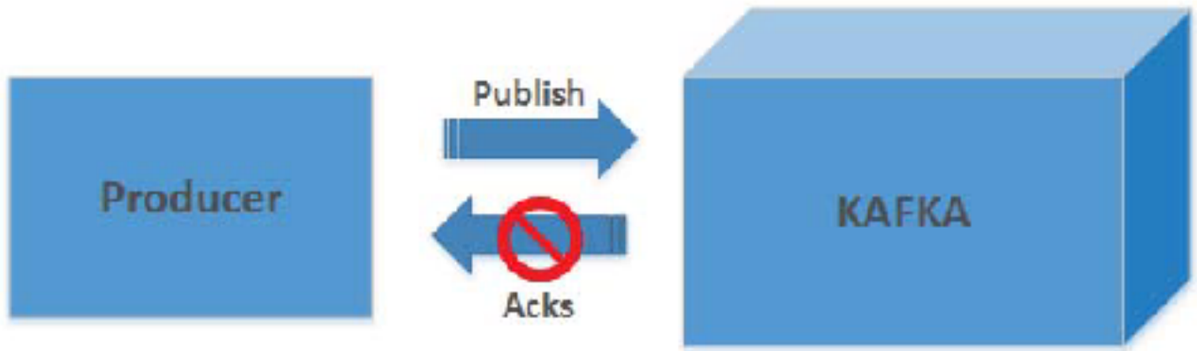
retires
If message sending fails, this represents the number of times producer will retry sending messages before it throws an exception.

partitioner.class
If you want to use a custom partitioner for your producer, then this configuration allows you to set the partitioner class, which implements the partitioner interface.

Common messaging publishing patterns

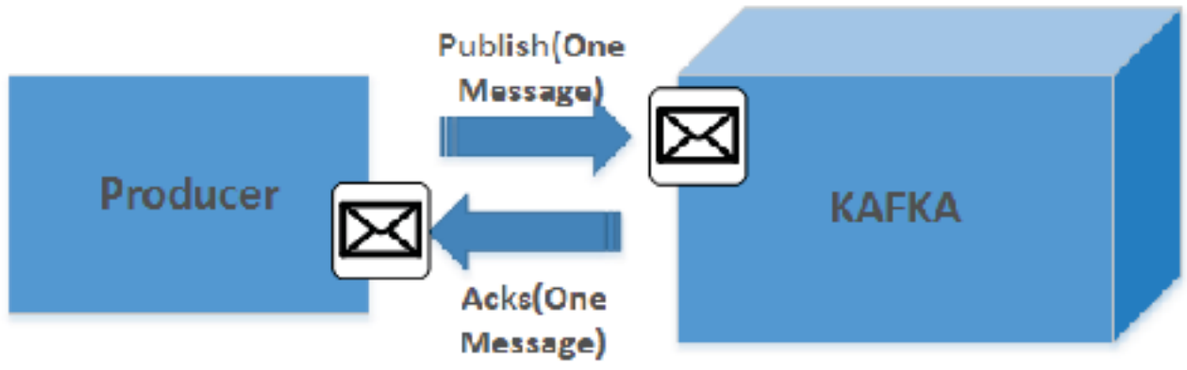
Fire-and-forget

To use the fire and forget model with Kafka, you have to set producer acks config to 0



One message transfers:

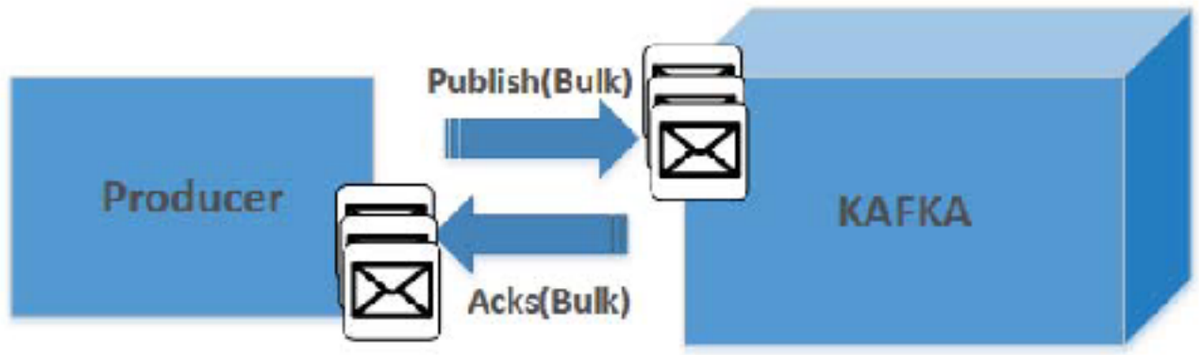
In this pattern, producer sends one message at a time. It can do so in synchronous or asynchronous mode.



APACHE kafka
A distributed streaming platform

Common messaging publishing patterns

Batching:
In this pattern, producers send multiple records to the same partition in a batch. The amount of memory required by a batch and wait time before sending the batch to Kafka is controlled by producer configuration parameters.



APACHE kafka
A distributed streaming platform

Best Practise

Data validation: One of the aspects that is usually forgotten while writing a producer system is to perform basic data validation tests on data that is to be written on the Kafka cluster. Some such examples could be conformity to schema, not null values for Key fields, and so on.

Exception handling: It is the sole responsibility of producer programs to decide on program flows with respect to exceptions. While writing a producer application, you should define different exception classes and as per your business requirements, decide on the actions that need to be taken

Number of retries: In general, there are two types of errors that you get in your producer application. The first type are errors that producer can retry, such as network timeouts and leader not available. The second type are errors that need to be handled by producer programs as mentioned in the preceding section. Configuring the number of retries will help you in mitigating risks related to message losses due to Kafka cluster errors or network errors.

Best Practise

Number of bootstrap URLs: You should always have more than one broker listed in your bootstrap broker configuration of your producer program. This helps producers to adjust to failures because if one of the brokers is not available, producers try to use all the listed brokers until it finds the one it can connect to

Avoid poor partitioning mechanism: Partitions are a unit of parallelism in Kafka. You should always choose an appropriate partitioning strategy to ensure that messages are distributed uniformly across all topic partitions.Poor partitioning strategy may lead to non-uniform message distribution and you would not be able to achieve the optimum parallelism out of your Kafka cluster. This is important in cases where you have chosen to use keys in your messages.

Temporary persistence of messages: For highly reliable systems, you should persist messages that are passing through your producer applications. Persistence could be on disk or in some kind of database. Persistence helps you replay messages in case of application failure or in case the Kafka cluster is unavailable due to some maintenance. This again, should be decided based on enterprise application requirements.

Best Practise

Avoid adding new partitions to existing topics: You should avoid adding partitions to existing topics when you are using key-based partitioning for message distribution.
Adding new partitions would change the calculated hash
code for each key as it takes the number of partitions as one of the inputs. You would end up having different partitions for the same key.

Kafka Consumer

The responsibilities of Kafka consumers apart from consuming messages from Kafka queues.

Subscribing to a topic: Consumer operations start with subscribing to a topic. If consumer is part of a consumer group, it will be assigned a subset of partitions from that topic. Consumer process would eventually read data from those assigned partitions

Consumer offset position: Kafka, unlike any other queues, does not maintain message offsets. Every consumer is responsible for maintaining its own consumer offset. Consumer offsets are maintained by consumer APIs and you do not have to do any additional coding for this

Replay / rewind / skip messages: Kafka consumer has full control over starting offsets to read messages from a topic partition. Using consumer APIs, any consumer application can pass the starting offsets to read messages from topic partitions. They can choose to read messages from the beginning or from some specific integer offset value irrespective of what the current offset value of a partition is.

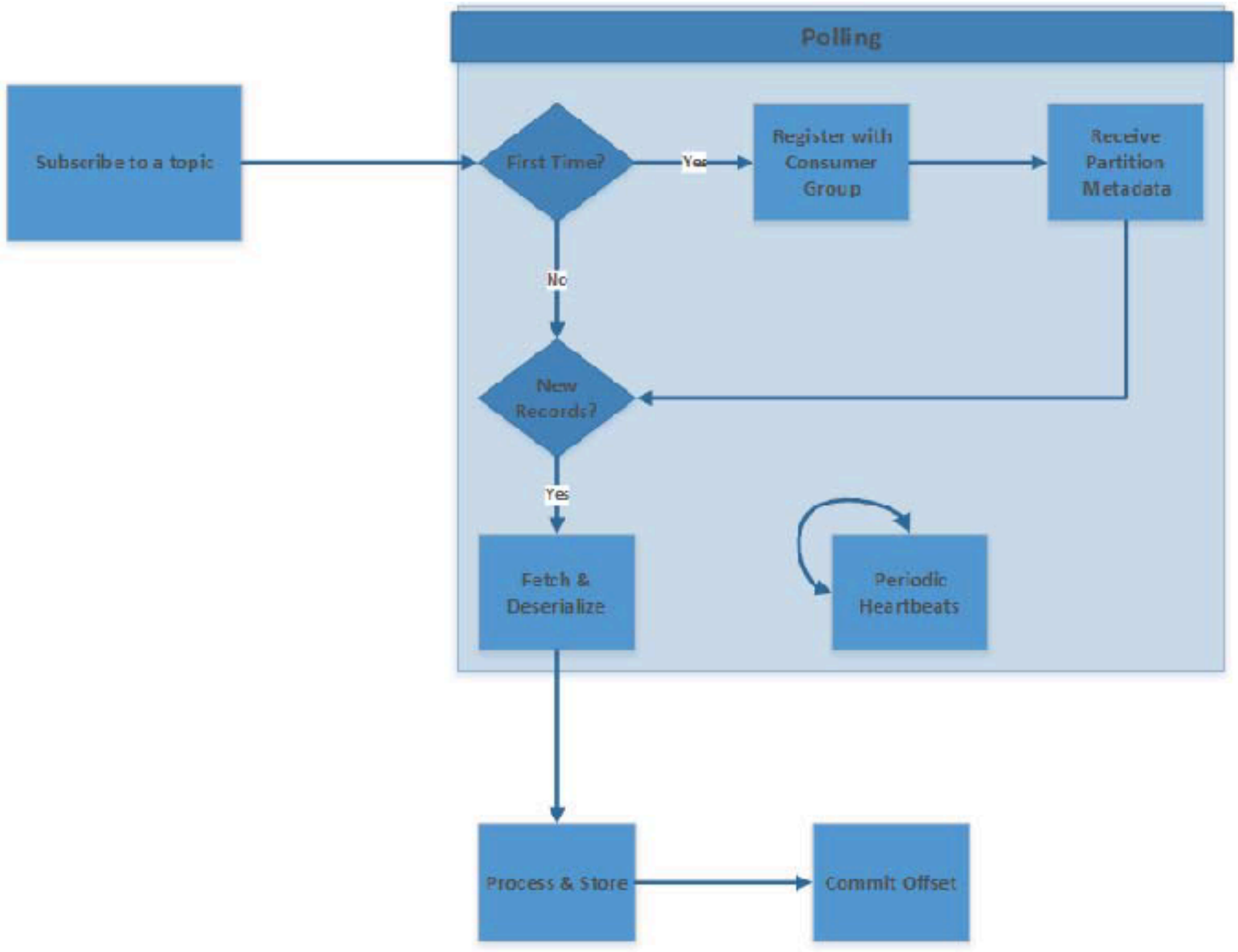
Kafka Consumer

Heartbeats: It is the consumer's responsibility to ensure that it sends regular heartbeat signals to the Kafka broker (consumer group leader) to confirm their membership and ownership of designated partitions. If heartbeats are not received by the group leader in a certain time interval, then the partition's ownership would be reassigned to some other consumer in the consumer group

Offset commits: Kafka does not track positions or offsets of the messages that are read from consumer applications. It is the responsibility of the consumer application to track their partition offset and commit it. This improves broker performance as they do not have to track each consumer offset and this gives flexibility to consumer applications in managing their offsets as per their specific scenarios. They can commit offsets after they finish processing a batch or they can commit offsets in the middle of very large batch processing to reduce side-effects of rebalancing.

Deserialization: Kafka producers serialize objects into byte arrays before they are sent to Kafka. Similarly, Kafka consumers deserialize these Java objects into byte arrays. Kafka consumer uses the deserializers that are the same as serializers used in the producer application.

Kafka Consumer



APACHE kafka®
A distributed streaming platform

Kafka Consumer API

Kafka also provides a rich set of APIs to develop a consumer application
There are basically **four** properties:

boostrap.servers : Deep Dive into Kafka Producers, for producer configuration. It takes a list of Kafka brokers' IPs.

key.deserializer: The difference is that in producer, we specified the class that can serialize the key of the message. Serialize means converting a key to a ByteArray. In consumer, we specify the class that can deserialize the ByteArray to a specific key type

value.deserializer: This property is used to deserialize the message. We should make sure that the deserializer class should match with the serializer class used to produce the data;

group.id: This property is not mandatory for the creation of a property but recommended to use while creating

Kafka Consumer Committing and polling

Polling is fetching data from the Kafka topic. Kafka returns the messages that have not yet been read by consumer.

How does Kafka know that consumer hasn't read the messages yet?

Consumer needs to tell Kafka that it needs data from a particular offset and therefore, consumer needs to store the latest read message somewhere so that in case of consumer failure, consumer can start reading from the next offset Kafka commits the offset of messages that it reads successfully

Auto commit: This is the default configuration of consumer. Consumer autocommits the offset of the latest read messages at the configured interval of time.
There are certain risks associated with this option.

For example, you set the interval to 10 seconds and consumer starts consuming the data. At the seventh second, your consumer fails, what will happen?
Consumer hasn’t committed the read offset yet so when it starts again, it will start reading from the start of the last committed offset and this will lead to duplicates.

Kafka Consumer Committing and polling

Current offset commit: Most of the time, we may want to have control over committing an offset when required. Kafka provides you with an API to enable this feature. It would be better to use this method call after we process all instances of ConsumerRecord, otherwise there is a risk of losing records if consumer fails in between.

Asynchronous commit: The problem with synchronous commit is that unless we receive an acknowledgment for a commit offset request from the Kafka server, consumer will be blocked. This will cost low throughput. It can be done by making commit happen asynchronously.
However, there is a problem in asynchronous commit. it may lead to duplicate message processing in a few cases where the order of the commit offset changes.

For example, offset of message 10 got committed before offset of message 5. In this case, K will again serve message 5-10 to consumer as the latest offset 10 is overridden by 5.

Additional consumer configuration

There are other optional configuration properties available for Kafka consumer that can play an important role in performance, memory, reliability

enable.auto.commit : If this is configured to true, then consumer will automatically commit the message offset after the configured interval of time.

fetch.min.bytes: This is the minimum amount of data in bytes that the Kafka server needs to return for a fetch request. In case the data is less than the configured number of bytes, the server will wait for enough data to accumulate and then send it to consumer.
Setting the value greater than the default, that is, one byte, will increase server throughpu will reduce latency of the consumer application.

request.timeout.ms This is the maximum amount of time that consumer will wait for a response to the request made before resending the request or failing when the maximum number of retries is reached

auto.offeset.reset This property is used when consumer doesn't have a valid offset for the partition from which it is reading the value.

Additional consumer configuration

latest: This value, if set to latest, means that the consumer will start reading from the latest message from the partition available at that
time when consumer started.

earliest: This value, if set to earliest, means that the consumer will start reading data from the beginning of the partition, which means that it will read all the data from the partition.

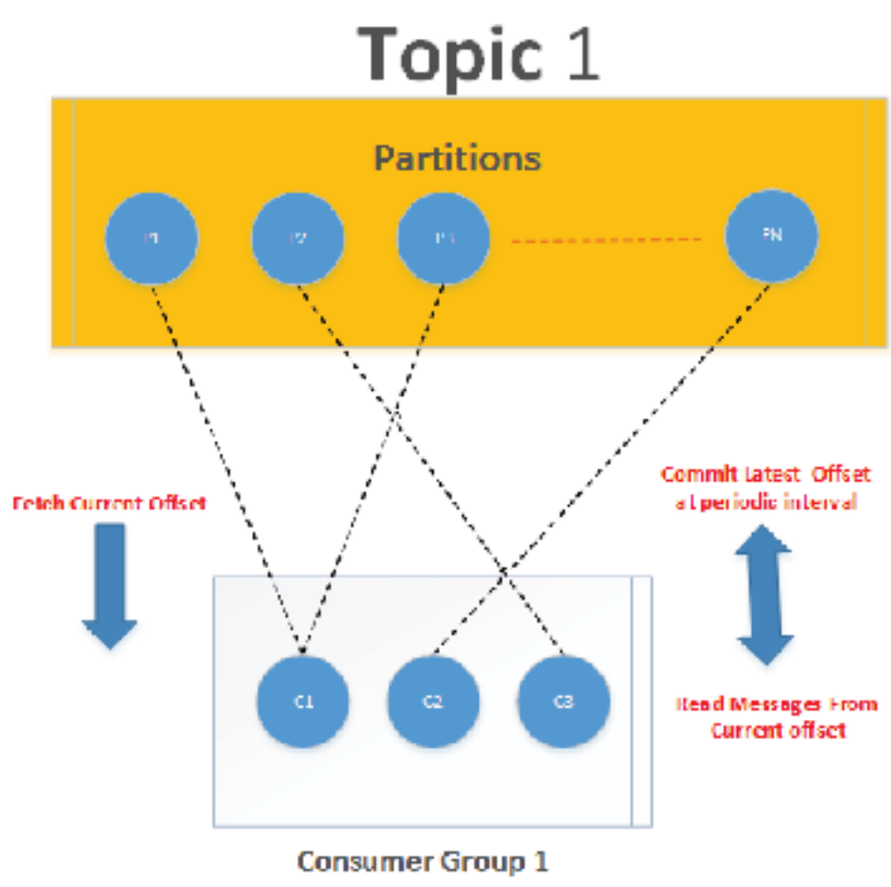
none: This value, if set to none, means that an exception will be thrown to the consumer.

session.timeout.ms Consumer sends a heartbeat to the consumer group coordinator to tell it that it is alive and restrict triggering the rebalancer. The consumer has to send heartbeats within the configured period of time.

max.partition.fetch.bytes This represents the maximum amount of data that the server will return per partition

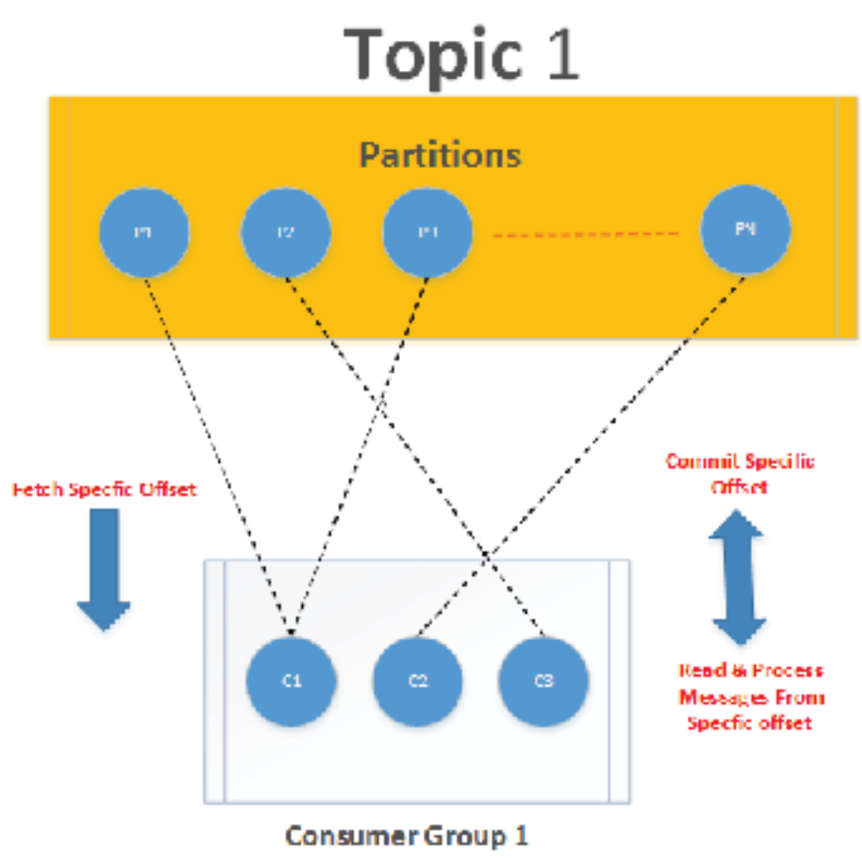
Common message consuming patterns

Consumer group - continuous data processing: In this pattern, once consumer is created and subscribes to a topic, it starts receiving messages from the current offset. The consumer commits the latest offsets based on the count of messages received in a batch at a regular, configured interval. The consumer checks whether it's time to commit, and if it is, it will commit the offsets. Offset commit can happen synchronously or asynchronously. It uses the auto-commit feature of the consumer API.



Common message consuming patterns

Consumer group - discrete data processing: Sometimes you want more control over consuming messages from Kafka. You want to read specific offsets of messages that may or may not be the latest current offset of the particular partition. Subsequently may want to commit specific offsets and not the regular latest offsets. This pattern outlines such a type of discrete data processing. In this, consumers fetch data based on the offset provided by them and they commit specific offsets that are as per their specific application requirements



Best Practise

Exception handling: Just like producers, it is the sole responsibility of consumer programs to decide on program flows with respect to exceptions. A consumer application should define different exception classes and, as per your business requirements, decide on the actions that need to be taken.

Handling rebalances: Whenever any new consumer joins consumer groups or any old consumer shuts down, a partition rebalance is triggered. Whenever a consumer is losing its partition ownership, it is imperative that they should commit the offsets of the last event that they have received from Kafka. For example, they should process and commit any in-memory buffered datasets before losing the ownership of a partition. Similarly, they should close any open file handles and database connection objects.

Best Practise

Commit offsets at the right time: If you are choosing to commit offset for messages, you need to do it at the right time. An application processing a batch of messages from Kafka may take more time to complete the processing of an entire batch; this is not a rule of thumb but if the processing time is more than a minute, try to commit the offset at regular intervals to avoid duplicate data processing in case the application fails. For more critical applications where processing duplicate data can cause huge costs, the commit offset time should be as short as possible if throughput is not an important factor.

Automatic offset commits: Choosing an auto-commit is also an option to go with where we do not care about processing duplicate records or want consumer to take care of the offset commit automatically. For example, the auto-commit interval is 10 seconds and at the seventh second, consumer fails. In this case, the offset for those seven seconds has not been committed and the next time the consumer recovers from failure, it will again process those seven seconds records.

Last