aiokafka Documentation

Release 0.8.0

Jun 01, 2023

CONTENTS

aiokafka is a client for the Apache Kafka distributed stream processing system using asyncio. It is based on the kafka-python library and reuses its internals for protocol parsing, errors, etc. The client is designed to function much like the official Java client, with a sprinkling of Pythonic interfaces.

aiokafka can be used with 0.9+ Kafka brokers and supports fully coordinated consumer groups – i.e., dynamic partition assignment to multiple consumers in the same group.

CHAPTER

ONE

GETTING STARTED

1.1 AIOKafkaConsumer

AIOKafkaConsumer is a high-level message consumer, intended to operate as similarly as possible to the official Java client.

Here's a consumer example:

```
from aiokafka import AIOKafkaConsumer
import asyncio

async def consume():
    consumer = AIOKafkaConsumer(
        'my_topic', 'my_other_topic',
        bootstrap_servers='localhost:9092',
        group_id="my-group")
    # Get cluster layout and join group `my-group`
    await consumer.start()
    try:
        # Consume messages
        async for msg in consumer:
            print("consumed: ", msg.topic, msg.partition, msg.offset,
                  msg.key, msg.value, msg.timestamp)
    finally:
        # Will leave consumer group; perform autocommit if enabled.
        await consumer.stop()

asyncio.run(consume())
```

Read more in *Consumer client* section.

1.2 AIOKafkaProducer

AIOKafkaProducer is a high-level, asynchronous message producer.

Here's a producer example:

```
from aiokafka import AIOKafkaProducer
import asyncio
```

(continues on next page)

(continued from previous page)

```
async def send_one():
    producer = AIOKafkaProducer(
        bootstrap_servers='localhost:9092')
    # Get cluster layout and initial topic/partition leadership information
    await producer.start()
    try:
        # Produce message
        await producer.send_and_wait("my_topic", b"Super message")
    finally:
        # Wait for all pending messages to be delivered or expire.
        await producer.stop()

asyncio.run(send_one())
```

Read more in *Producer client* section.

CHAPTER

TWO

INSTALLATION

```
pip install aiokafka
```

Note: **aiokafka** requires the kafka-python library.

2.1 Optional LZ4 install

To enable LZ4 compression/decompression, install **aiokafka** with `lz4` extra option:

> pip install 'aiokafka[lz4]'

Note, that on **Windows** you will need Visual Studio build tools, available for download from http://landinghub.visualstudio.com/visual-cpp-build-tools

2.2 Optional Snappy install

1. Download and build Snappy from http://google.github.io/snappy/

Ubuntu:

```
apt-get install libsnappy-dev
```

OSX:

```
brew install snappy
```

From Source:

```
wget https://github.com/google/snappy/tarball/master
tar xzvf google-snappy-X.X.X-X-XXXXXXXX.tar.gz
cd google-snappy-X.X.X-X-XXXXXXXX
./configure
make
sudo make install
```

1. Install **aiokafka** with snappy extra option

```
pip install 'aiokafka[snappy]'
```

2.3 Optional zstd indtall

To enable Zstandard compression/decompression, install **aiokafka** with `zstd` extra option:

```
pip install 'aiokafka[zstd]'
```

2.4 Optional GSSAPI install

To enable SASL authentication with GSSAPI, install **aiokafka** with `gssapi` extra option:

```
pip install 'aiokafka[gssapi]'
```

CHAPTER

THREE

SOURCE CODE

The project is hosted on GitHub

Please feel free to file an issue on bug tracker if you have found a bug or have some suggestion for library improvement.

The library uses Travis for Continious Integration.

CHAPTER

FOUR

AUTHORS AND LICENSE

The **aiokafka** package is Apache 2 licensed and freely available.

Feel free to improve this package and send a pull request to GitHub.

Contents:

4.1 Producer client

AIOKafkaProducer is a client that publishes records to the Kafka cluster. Most simple usage would be:

```
producer = aiokafka.AIOKafkaProducer(bootstrap_servers="localhost:9092")
await producer.start()
try:
    await producer.send_and_wait("my_topic", b"Super message")
finally:
    await producer.stop()
```

Under the hood, *AIOKafkaProducer* does quite some work on message delivery including batching, retries, etc. All of it can be configured, so let's go through some components for a better understanding of the configuration options.

4.1.1 Message buffering

While the user would expect the example above to send `"Super message"` directly to the broker, it's actually not sent right away, but rather added to a **buffer space**. A background task will then get batches of messages and send them to appropriate nodes in the cluster. This batching scheme allows *more throughput and more efficient compression*. To see it more clearly lets avoid the *send_and_wait()* shortcut:

```
# Will add the message to 1st partition's batch. If this method times out,
# we can say for sure that message will never be sent.

fut = await producer.send("my_topic", b"Super message", partition=1)

# Message will either be delivered or an unrecoverable error will occur.
# Cancelling this future will not cancel the send.
msg = await fut
```

Batches themselves are created **per partition** with a maximum size of *max_batch_size*. Messages in a batch are strictly in append order and only 1 batch per partition is sent at a time (**aiokafka** does not support `max.inflight.requests.per.connection` option present in Java client). This makes a strict guarantee on message order in a partition.

By default, a new batch is sent immediately after the previous one (even if it's not full). If you want to reduce the number of requests you can set *linger_ms* to something other than 0. This will add an additional delay before sending next batch if it's not yet full.

aiokafka does not (yet!) support some options, supported by Java's client:

- `buffer.memory` to limit how much buffer space is used by Producer to schedule requests in *all partitions*.
- `max.block.ms` to limit the amount of time *send()* coroutine will wait for buffer append when the memory limit is reached. For now use:

```
await asyncio.wait_for(producer.send(...), timeout=timeout)
```

If your use case requires direct batching control, see *Direct batch control*.

4.1.2 Retries and Message acknowledgement

aiokafka will retry most errors automatically, but only until *request_timeout_ms*. If a request is expired, the last error will be raised to the application. Retrying messages on application level after an error will potentially lead to duplicates, so it's up to the user to decide.

For example, if *RequestTimedOutError* is raised, Producer can't be sure if the Broker wrote the request or not.

The *acks* option controls when the produce request is considered acknowledged.

The most durable setting is `acks="all"`. Broker will wait for all available replicas to write the request before replying to Producer. Broker will consult it's `min.insync.replicas` setting to know the minimal amount of replicas to write. If there's not enough in sync replicas either *NotEnoughReplicasError* or *NotEnoughReplicasAfterAppendError* will be raised. It's up to the user what to do in those cases, as the errors are not retriable.

The default is `ack=1` setting. It will not wait for replica writes, only for Leader to write the request.

The least safe is `ack=0` when there will be no acknowledgement from Broker, meaning client will never retry, as it will never see any errors.

4.1.3 Idempotent produce

As of Kafka 0.11 the Brokers support idempotent producing, that will prevent the Producer from creating duplicates on retries. *aiokafka* supports this mode by passing the parameter `enable_idempotence=True` to *AIOKafkaProducer*:

```
producer = aiokafka.AIOKafkaProducer(
    bootstrap_servers='localhost:9092',
    enable_idempotence=True)
await producer.start()
try:
    await producer.send_and_wait("my_topic", b"Super message")
finally:
    await producer.stop()
```

This option will change a bit the logic on message delivery:

- The above mentioned `ack="all"` will be forced. If any other value is explicitly passed with `enable_idempotence=True` a `ValueError` will be raised.
- I contrast to general mode, will not raise *RequestTimedOutError* errors and will not expire batch delivery after `request_timeout_ms` passed.

New in version 0.5.0.

4.1.4 Transactional producer

As of Kafka 0.11 the Brokers support transactional message producer, meaning that messages sent to one or more topics will only be visible on consumers after the transaction is committed. To use the transactional producer and the attendant APIs, you must set the `transactional_id` configuration property:

```
producer = aiokafka.AIOKafkaProducer(
    bootstrap_servers='localhost:9092',
    transactional_id="transactional_test")
await producer.start()
try:
    async with producer.transaction():
        res = await producer.send_and_wait(
            "test-topic", b"Super transactional message")
finally:
    await producer.stop()
```

If the *transactional_id* is set, idempotence is automatically enabled along with the producer configs which idempotence depends on. Further, topics which are included in transactions should be configured for durability. In particular, the `replication.factor` should be at least 3, and the `min.insync.replicas` for these topics should be set to 2. Finally, in order for transactional guarantees to be realized from end-to-end, the consumers must be configured to read only committed messages as well. See *Reading Transactional Messages*.

The purpose of the *transactional_id* is to enable transaction recovery across multiple sessions of a single producer instance. It would typically be derived from the shard identifier in a partitioned, stateful, application. As such, it should be unique to each producer instance running within a partitioned application. Using the same *transactional_id* will cause the previous instance to raise an exception `ProducerFenced` that is not retriable and will force it to exit.

Besides, the `transaction()` shortcut producer also supports a set of API's similar to ones in Java Client. See the `AIOKafkaProducer` API docs.

Besides being able to commit several topics atomically, as offsets are also stored in a separate system topic it's possible to commit a consumer offset as part of the same transaction:

```
async with producer.transaction():
    commit_offsets = {
        TopicPartition("some-topic", 0): 100
    }
    await producer.send_offsets_to_transaction(
        commit_offsets, "some-consumer-group")
```

See a more full example in *Transactional Consume-Process-Produce*.

New in version 0.5.0.

4.1.5 Returned `RecordMetadata` object

After a message is sent, the user receives a `RecordMetadata` object.

Note: In a very rare case, when Idempotent or Transactional producer is used and there was a long wait between batch initial send and a retry, producer may return `offset == -1` and `timestamp == -1` as Broker already expired the metadata for this produce sequence and only knows that it's a duplicate due to a larger sequence present

4.1.6 Direct batch control

Users who need precise control over batch flow may use the lower-level *create_batch()* and *send_batch()* interfaces:

```
# Create the batch without queueing for delivery.
batch = producer.create_batch()

# Populate the batch. The append() method will return metadata for the
# added message or None if batch is full.
for i in range(2):
    metadata = batch.append(value=b"msg %d" % i, key=None, timestamp=None)
    assert metadata is not None

# Optionally close the batch to further submission. If left open, the batch
# may be appended to by producer.send().
batch.close()

# Add the batch to the first partition's submission queue. If this method
# times out, we can say for sure that batch will never be sent.
fut = await producer.send_batch(batch, "my_topic", partition=1)

# Batch will either be delivered or an unrecoverable error will occur.
# Cancelling this future will not cancel the send.
record = await fut
```

While any number of batches may be created, only a single batch per partition is sent at a time. Additional calls to *send_batch()* against the same partition will wait for the inflight batch to be delivered before sending.

Upon delivery, the `record`'s *offset* will match the batch's first message.

4.2 Consumer client

AIOKafkaConsumer is a client that consumes records from a Kafka cluster. Most simple usage would be:

```
consumer = aiokafka.AIOKafkaConsumer(
    "my_topic",
    bootstrap_servers='localhost:9092'
)
await consumer.start()
try:
    async for msg in consumer:
        print(
            "{}:{:d}:{:d}: key={} value={} timestamp_ms={}".format(
                msg.topic, msg.partition, msg.offset, msg.key, msg.value,
                msg.timestamp)
        )
finally:
    await consumer.stop()
```

Note: *msg.value* and *msg.key* are raw bytes, use *AIOKafkaConsumer*'s *key_deserializer* and *value_deserializer* configuration if you need to decode them.

Note: `AIOKafkaConsumer` maintains TCP connections as well as a few background tasks to fetch data and coordinate assignments. Failure to call `AIOKafkaConsumer.stop()` after consumer use *will leave background tasks running*.

`AIOKafkaConsumer` transparently handles the failure of Kafka brokers and transparently adapts as topic partitions it fetches migrate within the cluster. It also interacts with the broker to allow groups of consumers to load balance consumption using *Consumer Groups*.

4.2.1 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:

```
msg = await consumer.getone()
print(msg.offset)  # Unique msg autoincrement ID in this topic-partition.

tp = aiokafka.TopicPartition(msg.topic, msg.partition)

position = await consumer.position(tp)
# Position is the next fetched offset
assert position == msg.offset + 1

committed = await consumer.committed(tp)
print(committed)
```

Note: To use the `commit()` and `committed()` APIs you need to set `group_id` to something other than `None`. See *consumer-groups* below.

Here if the consumer is at *position* `5`, it has consumed records with *offsets* `0` through `4` and will next receive the record with *offset* `5`.

There are actually two *notions of position*:

- The *position* gives the *offset* of the next record that should be given out. It will be *one larger* than the highest *offset* the consumer has seen in that partition. It automatically increases every time the consumer yields messages in either `getmany()` or `getone()` calls.
- The *committed position* is the last *offset* that has been stored securely. Should the process restart, this is the offset that the consumer will start from. The consumer can either *automatically commit offsets periodically*, or it can choose to control this committed position *manually* by calling `await consumer.commit()`.

This distinction gives the consumer control over when a record is considered consumed. It is discussed in further detail below.

Manual vs automatic committing

For most simple use cases auto committing is probably the best choice:

```
consumer = AIOKafkaConsumer(
    "my_topic",
    bootstrap_servers='localhost:9092',
    group_id="my_group",           # Consumer must be in a group to commit
    enable_auto_commit=True,       # Is True by default anyway
    auto_commit_interval_ms=1000,  # Autocommit every second
    auto_offset_reset="earliest",  # If committed offset not found, start
                                   # from beginning
)
await consumer.start()

async for msg in consumer:  # Will periodically commit returned messages.
    # process message
    pass
```

This example can have *"At least once"* delivery semantics, but only if we process messages **one at a time**. If you want *"At least once"* semantics for batch operations you should use *manual commit*:

```
consumer = AIOKafkaConsumer(
    "my_topic",
    bootstrap_servers='localhost:9092',
    group_id="my_group",           # Consumer must be in a group to commit
    enable_auto_commit=False,      # Will disable autocommit
    auto_offset_reset="earliest",  # If committed offset not found, start
                                   # from beginning
)
await consumer.start()

batch = []
async for msg in consumer:
    batch.append(msg)
    if len(batch) == 100:
        await process_msg_batch(batch)
        await consumer.commit()
        batch = []
```

> **Warning:** When using **manual commit** it is recommended to provide a `ConsumerRebalanceListener` which will process pending messages in the batch and commit before allowing rejoin. If your group will rebalance during processing commit will fail with `CommitFailedError`, as partitions may have been processed by other consumer already.

This example will hold on to messages until we have enough to process in bulk. The algorithm can be enhanced by taking advantage of:

- `await consumer.getmany()` to avoid multiple calls to get a batch of messages.
- `await consumer.highwater(partition)` to understand if we have more unconsumed messages or this one is the last one in the partition.

If you want to have more control over which partition and message is committed, you can specify offset manually:

```
while True:
    result = await consumer.getmany(timeout_ms=10 * 1000)
    for tp, messages in result.items():
        if messages:
            await process_msg_batch(messages)
            # Commit progress only for this partition
            await consumer.commit({tp: messages[-1].offset + 1})
```

Note: The committed offset should always be the offset of the next message that your application will read. Thus, when calling *await consumer.commit(offset)* you should add one to the offset of the last message processed.

Here we process a batch of messages per partition and commit not all consumed *offsets*, but only for the partition, we processed.

Controlling The Consumer's Position

In most use cases the consumer will simply consume records from beginning to end, periodically committing its position (either automatically or manually). If you only want your consumer to process newest messages, you can ask it to start from `latest` offset:

```
consumer = AIOKafkaConsumer(
    "my_topic",
    bootstrap_servers='localhost:9092',
    auto_offset_reset="latest",
)
await consumer.start()

async for msg in consumer:
    # process message
    pass
```

Note: If you have a valid **committed position** consumer will use that. `auto_offset_reset` will only be used when the position is invalid.

Kafka also allows the consumer to manually control its position, moving forward or backwards in a partition at will using *AIOKafkaConsumer.seek()*. For example, you can re-consume records:

```
msg = await consumer.getone()
tp = TopicPartition(msg.topic, msg.partition)

consumer.seek(tp, msg.offset)
msg2 = await consumer.getone()

assert msg2 == msg
```

Also you can combine it with `offset_for_times` API to query to specific offsets based on timestamp.

There are several use cases where manually controlling the consumer's position can be useful.

One case is for **time-sensitive record processing** it may make sense for a consumer that falls far enough behind to not attempt to catch up processing all records, but rather just skip to the most recent records. Or you can use `offsets_for_times` API to get the offsets after certain timestamp.

Another use case is for a **system that maintains local state**. In such a system the consumer will want to initialize its position on startup to whatever is contained in the local store. Likewise, if the local state is destroyed (say because the disk is lost) the state may be recreated on a new machine by re-consuming all the data and recreating the state (assuming that Kafka is retaining sufficient history).

See also related configuration params and API docs:

- *auto_offset_reset* config option to set behaviour in case the position is either undefined or incorrect.
- `seek()`, `seek_to_beginning()`, `seek_to_end()` API's to force position change on partition('s).
- `offsets_for_times()`, `beginning_offsets()`, `end_offsets()` API's to query offsets for partitions even if they are not assigned to this consumer.

Storing Offsets Outside Kafka

Storing *offsets* in Kafka is optional, you can store offsets in another place and use `seek()` API to start from saved position. The primary use case for this is allowing the application to store both the offset and the results of the consumption in the same system in a way that both the results and offsets are stored atomically. For example, if we save aggregated by *key* counts in Redis:

```
import json
from collections import Counter

redis = await aioredis.create_redis(("localhost", 6379))
REDIS_HASH_KEY = "aggregated_count:my_topic:0"

tp = TopicPartition("my_topic", 0)
consumer = AIOKafkaConsumer(
    bootstrap_servers='localhost:9092',
    enable_auto_commit=False,
)
await consumer.start()
consumer.assign([tp])

# Load initial state of aggregation and last processed offset
offset = -1
counts = Counter()
initial_counts = await redis.hgetall(REDIS_HASH_KEY, encoding="utf-8")
for key, state in initial_counts.items():
    state = json.loads(state)
    offset = max([offset, state['offset']])
    counts[key] = state['count']

# Same as with manual commit, you need to fetch next message, so +1
consumer.seek(tp, offset + 1)

async for msg in consumer:
    key = msg.key.decode("utf-8")
    counts[key] += 1
    value = json.dumps({
        "count": counts[key],
        "offset": msg.offset
```

(continues on next page)

(continued from previous page)

```
    })
    await redis.hset(REDIS_HASH_KEY, key, value)
```

So to save results outside of Kafka you need to:

- Configure: `enable.auto.commit=false`
- Use the offset provided with each `ConsumerRecord` to save your position
- On restart or rebalance restore the position of the consumer using `seek()`

This is not always possible, but when it is it will make the consumption fully atomic and give *exactly once* semantics that are stronger than the default *at-least once* semantics you get with Kafka's offset commit functionality.

This type of usage is simplest when the partition assignment is also done manually (like we did above). If the partition assignment is done automatically special care is needed to handle the case where partition assignments change. See *Local state and storing offsets outside of Kafka* example for more details.

4.2.2 Consumer Groups and Topic Subscriptions

Kafka uses the concept of **Consumer Groups** to allow a pool of processes to divide the work of consuming and processing records. These processes can either be running on the same machine or they can be distributed over many machines to provide scalability and fault tolerance for processing.

All `AIOKafkaConsumer` instances sharing the same `group_id` will be part of the same **Consumer Group**:

```
# Process 1
consumer = AIOKafkaConsumer(
    "my_topic", bootstrap_servers='localhost:9092',
    group_id="MyGreatConsumerGroup"  # This will enable Consumer Groups
)
await consumer.start()
async for msg in consumer:
    print("Process %s consumed msg from partition %s" % (
          os.getpid(), msg.partition))

# Process 2
consumer2 = AIOKafkaConsumer(
    "my_topic", bootstrap_servers='localhost:9092',
    group_id="MyGreatConsumerGroup"  # This will enable Consumer Groups
)
await consumer2.start()
async for msg in consumer2:
    print("Process %s consumed msg from partition %s" % (
          os.getpid(), msg.partition))
```

Each consumer in a group can dynamically set the list of topics it wants to subscribe to through `subscribe()` call. Kafka will deliver each message in the subscribed topics to only one of the processes 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.

Membership in a consumer group is maintained dynamically: if a process fails, the partitions assigned to it *will be reassigned to other consumers* in the same group. Similarly, if a new consumer joins the group, partitions will be *moved from existing consumers to the new one*. This is known as **rebalancing the group**.

Note: Conceptually you can think of a **Consumer Group** as being a *single logical subscriber* that happens to be made up of multiple processes.

In addition, when group reassignment happens automatically, consumers can be notified through a `ConsumerRebalanceListener`, which allows them to finish necessary application-level logic such as state cleanup, manual offset commits, etc. See `subscribe()` docs for more details.

Warning: Be careful with `ConsumerRebalanceListener` to avoid deadlocks. The Consumer will await the defined handlers and will block subsequent calls to `getmany()` and `getone()`. For example this code will deadlock:

```
lock = asyncio.Lock()
consumer = AIOKafkaConsumer(...)

class MyRebalancer(aiokafka.ConsumerRebalanceListener):

    async def on_partitions_revoked(self, revoked):
        async with lock:
            pass

    async def on_partitions_assigned(self, assigned):
        pass

async def main():
    consumer.subscribe("topic", listener=MyRebalancer())
    while True:
        async with lock:
            msgs = await consumer.getmany(timeout_ms=1000)
            # process messages
```

You need to put `consumer.getmany(timeout_ms=1000)` call outside of the lock.

For more information on how **Consumer Groups** are organized see Official Kafka Docs.

Topic subscription by pattern

`AIOKafkaConsumer` performs periodic metadata refreshes in the background and will notice when new partitions are added to one of the subscribed topics or when a new topic matching a *subscribed regex* is created. For example:

```
consumer = AIOKafkaConsumer(
    bootstrap_servers='localhost:9092',
    metadata_max_age_ms=30000,  # This controls the polling interval
)
await consumer.start()
consumer.subscribe(pattern="^MyGreatTopic-.*$")

async for msg in consumer:  # Will detect metadata changes
    print("Consumed msg %s %s %s" % (msg.topic, msg.partition, msg.value))
```

Here, the consumer will automatically detect new topics like `MyGreatTopic-1` or `MyGreatTopic-2` and start consuming them.

If you use Consumer Groups the group's *Leader* will trigger a **group rebalance** when it notices metadata changes. It's because only the *Leader* has full knowledge of which topics are assigned to the group.

Manual partition assignment

It is also possible for the consumer to manually assign specific partitions using *assign([tp1, tp2])*. In this case, dynamic partition assignment and consumer group coordination will be disabled. For example:

```
consumer = AIOKafkaConsumer(
    bootstrap_servers='localhost:9092'
)
tp1 = TopicPartition("my_topic", 1)
tp2 = TopicPartition("my_topic", 2)
consumer.assign([tp1, tp2])

async for msg in consumer:
    print("Consumed msg %s %s %s", msg.topic, msg.partition, msg.value)
```

`group_id` can still be used for committing position, but be careful to avoid **collisions** with multiple instances sharing the same group.

It is not possible to mix manual partition assignment *assign()* and topic subscription *subscribe()*. An attempt to do so will result in an *IllegalStateError*.

Consumption Flow Control

By default Consumer will fetch from all partitions, effectively giving these partitions the same priority. However in some cases, you would want for some partitions to have higher priority (say they have more lag and you want to catch up). For example:

```
consumer = AIOKafkaConsumer("my_topic", ...)

partitions = []  # Fetch all partitions on first request
while True:
    msgs = await consumer.getmany(*partitions)
    # process messages
    await process_messages(msgs)

    # Prioritize partitions, that lag behind.
    partitions = []
    for partition in consumer.assignment():
        highwater = consumer.highwater(partition)
        position = await consumer.position(partition)
        position_lag = highwater - position
        timestamp = consumer.last_poll_timestamp(partition)
        time_lag = time.time() * 1000 - timestamp
        if position_lag > POSITION_THRESHOLD or time_lag > TIME_THRESHOLD:
            partitions.append(partition)
```

Note: This interface differs from *pause()* / *resume()* interface of kafka-python and Java clients.

Here we will consume all partitions if they do not lag behind, but if some go above a certain *threshold*, we will consume them to catch up. This can very well be used in a case where some consumer died and this consumer took over its partitions, that are now lagging behind.

Some things to note about it:

- There may be a slight **pause in consumption** if you change the partitions you are fetching. This can happen when Consumer requests a fetch for partitions that have no data available. Consider setting a relatively low `fetch_max_wait_ms` to avoid this.
- The `async for` interface can not be used with explicit partition filtering, just use `getone()` instead.

Reading Transactional Messages

Transactions were introduced in Kafka 0.11.0 wherein applications can write to multiple topics and partitions atomically. In order for this to work, consumers reading from these partitions should be configured to only read committed data. This can be achieved by by setting the `isolation_level=read_committed` in the consumer's configuration:

```
consumer = aiokafka.AIOKafkaConsumer(
    "my_topic",
    bootstrap_servers='localhost:9092',
    isolation_level="read_committed"
)
await consumer.start()
async for msg in consumer:  # Only read committed tranasctions
    pass
```

In *read_committed* mode, the consumer will read only those transactional messages which have been successfully committed. It will continue to read non-transactional messages as before. There is no client-side buffering in *read_committed* mode. Instead, the end offset of a partition for a *read_committed* consumer would be the offset of the first message in the partition belonging to an open transaction. This offset is known as the **Last Stable Offset** (LSO).

A *read_committed* consumer will only read up to the LSO and filter out any transactional messages which have been aborted. The LSO also affects the behavior of `seek_to_end()` and `end_offsets()` for *read_committed* consumers, details of which are in each method's documentation. Finally, `last_stable_offset()` API was added similarly to `highwater()` API to query the lSO on a currently assigned transaction:

```
async for msg in consumer:  # Only read committed tranasctions
    tp = TopicPartition(msg.topic, msg.partition)
    lso = consumer.last_stable_offset(tp)
    lag = lso - msg.offset
    print(f"Consumer is behind by {lag} messages")

    end_offsets = await consumer.end_offsets([tp])
    assert end_offsets[tp] == lso

await consumer.seek_to_end(tp)
position = await consumer.position(tp)
```

Partitions with transactional messages will include commit or abort markers which indicate the result of a transaction. There markers are not returned to applications, yet have an offset in the log. As a result, applications reading from topics with transactional messages will see gaps in the consumed offsets. These missing messages would be the transaction markers, and they are filtered out for consumers in both isolation levels. Additionally, applications using *read_committed* consumers may also see gaps due to aborted transactions, since those messages would not be returned by the consumer and yet would have valid offsets.

4.2.3 Detecting Consumer Failures

People who worked with kafka-python or Java Client probably know that the `poll()` API is designed to ensure liveness of a Consumer Groups. In other words, Consumer will only be considered alive if it consumes messages. It's not the same for **aiokafka**, for more details read *Difference between aiokafka and kafka-python*.

aiokafka will join the group on `start()` and will send heartbeats in the background, keeping the group alive, same as Java Client. But in the case of a rebalance it will also done in the background.

Offset commits in autocommit mode is done strictly by time in the background (in Java client autocommit will not be done if you don't call `poll()` another time).

4.3 Difference between aiokafka and kafka-python

4.3.1 Why do we need another library?

kafka-python is a great project, which tries to fully mimic the interface of the Java Client API. It is more *feature* oriented, rather than *speed*, but still gives quite good throughput. It's actively developed and is fast to react to changes in the Java client.

While kafka-python has a lot of great features it is made to be used in a **Threaded** environment. Even more, it mimics Java's client, making it **Java's threaded** environment, which does not have that much of *asynchronous* ways of doing things. It's not **bad** as Java's Threads are very powerful with the ability to use multiple cores.

The API itself just can't be adopted to be used in an asynchronous way (even though the library does asynchronous IO using *selectors*). It has too much blocking behavior including *blocking* socket usage, threading synchronization, etc. Examples would be:

- *bootstrap*, which blocks in the constructor itself
- blocking iterator for consumption
- sending produce requests block if buffer is full

All those can't be changed to use `Future` API seamlessly. So to get a normal, non-blocking interface based on `Future`'s and coroutines a new library needed to be written.

4.3.2 API differences and rationale

aiokafka has some differences in API design. While the **Producer** is mostly the same, **Consumer** has some significant differences, that we want to talk about.

Consumer has no `poll()` method

In kafka-python, `kafka.KafkaConsumer.poll()` is a blocking call that performs not only message fetching, but also:

- Socket polling using *epoll*, *kqueue* or other available API of your OS.
- Ensures liveliness of a Consumer Group
- Does autocommit

This will never be a case where you own the IO loop, at least not with socket polling. To avoid misunderstandings as to why do those methods behave in a different way `AIOKafkaConsumer` exposes this interface under the name `getmany()` with some other differences described below.

Rebalances are happening in the background

In original Kafka Java Client before 0.10.1 heartbeats were only sent if `poll()` was called. This lead to a lot of issues (reasons for KIP-41 and KIP-62 proposals) and workarounds using `pause()` and `poll(0)` for heartbeats. After Java client and kafka-python also changed the behaviour to a background Thread sending, that mitigated most issues.

aiokafka delegates heartbeating to a background *Task* and will send heartbeats to Coordinator as long as the *event loop* is running. This behaviour is very similar to Java client, with the exception of no heartbeats on long CPU bound methods.

But **aiokafka** also performs group rebalancing in the same background Task. This means, that the processing time between *`getmany()`* calls actually does not affect rebalancing. `KIP-62` proposed to provide `max.poll.interval.ms` as the configuration for both *rebalance timeout* and *consumer processing timeout*. In **aiokafka** it does not make much sense, as those 2 are not related, so we added both configurations (`rebalance_timeout_ms` and `max_poll_interval_ms`).

It is quite critical to provide *`ConsumerRebalanceListener`* if you need to control rebalance start and end moments. In that case set the `rebalance_timeout_ms` to the maximum time your application can spend waiting in the callback. If your callback waits for the last *`getmany()`* result to be processed, it is safe to set this value to `max_poll_interval_ms`, same as in Java client.

Prefetching is more sophisticated

In the Kafka Java Client and kafka-python, the prefetching is very simple, as it only performs prefetches:

- in `poll()` call if we don't have enough data stored to satisfy another `poll()`
- in the *iterator* interface if we have processed *nearly* all data.

A very simplified version would be:

```
def poll():
    max_records = self.config['max_poll_records']
    records = consumer.fethed_records(max_records)
    if not consumer.has_enough_records(max_records)
        consumer.send_fetches()  # prefetch another batch
    return records
```

This works great for throughput as the algorithm is simple and we pipeline IO task with record processing.

But it does not perform as great in case of **semantic partitioning**, where you may have per-partition processing. In this case latency will be bound to the time of processing of data in all topics.

Which is why **aiokafka** tries to do prefetches **per partition**. For example, if we processed all data pending for a partition in *iterator* interface, **aiokafka** will *try* to prefetch new data right away. The same interface could be built on top of kafka-python's `pause()` API, but would require a lot of code.

Note: Using *`getmany()`* without specifying partitions will result in the same prefetch behaviour as using `poll()`.

4.4 API Documentation

4.4.1 Producer class

class aiokafka.**AIOKafkaProducer**(**, loop=None, bootstrap_servers='localhost', client_id=None, metadata_max_age_ms=300000, request_timeout_ms=40000, api_version='auto', acks=<object object>, key_serializer=None, value_serializer=None, compression_type=None, max_batch_size=16384, partitioner=<kafka.partitioner.default.DefaultPartitioner object>, max_request_size=1048576, linger_ms=0, send_backoff_ms=100, retry_backoff_ms=100, security_protocol='PLAINTEXT', ssl_context=None, connections_max_idle_ms=540000, enable_idempotence=False, transactional_id=None, transaction_timeout_ms=60000, sasl_mechanism='PLAIN', sasl_plain_password=None, sasl_plain_username=None, sasl_kerberos_service_name='kafka', sasl_kerberos_domain_name=None, sasl_oauth_token_provider=None*)

A Kafka client that publishes records to the Kafka cluster.

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 task that is responsible for turning these records into requests and transmitting them to the cluster.

The `send()` 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.

The *acks* config controls the criteria under which requests are considered complete. The `all` setting will result in waiting for all replicas to respond, the slowest but most durable setting.

The *key_serializer* and *value_serializer* instruct how to turn the key and value objects the user provides into `bytes`.

Parameters

- **bootstrap_servers** (*str, list(str)*) – a `host[:port]` string or list of `host[:port]` strings that the producer should contact to bootstrap initial cluster metadata. This does not have to be the full node list. It just needs to have at least one broker that will respond to a Metadata API Request. Default port is 9092. If no servers are specified, will default to `localhost:9092`.
- **client_id** (*str*) – a name for this client. This string is passed in each request to servers and can be used to identify specific server-side log entries that correspond to this client. Default: `aiokafka-producer-#` (appended with a unique number per instance)
- **key_serializer** (*Callable*) – used to convert user-supplied keys to bytes If not `None`, called as `f(key),` should return `bytes`. Default: `None`.
- **value_serializer** (*Callable*) – used to convert user-supplied message values to `bytes`. If not `None`, called as `f(value)`, should return `bytes`. Default: `None`.
- **acks** (*Any*) – one of `0`, `1`, `all`. The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The following settings are common:
  - `0`: Producer will not wait for any acknowledgment from the server at all. The message will immediately be added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the retries configuration will

not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to -1.

  - 1: The broker leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost.
  - all: The broker leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.

  If unset, defaults to acks=1. If *enable_idempotence* is True defaults to acks=all

- **compression_type** (*str*) – The compression type for all data generated by the producer. Valid values are gzip, snappy, lz4, zstd or None. Compression is of full batches of data, so the efficacy of batching will also impact the compression ratio (more batching means better compression). Default: None.
- **max_batch_size** (*int*) – Maximum size of buffered data per partition. After this amount *send()* coroutine will block until batch is drained. Default: 16384
- **linger_ms** (*int*) – The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding a small amount of artificial delay; that is, if first request is processed faster, than *linger_ms*, producer will wait linger_ms - process_time. Default: 0 (i.e. no delay).
- **partitioner** (*Callable*) – Callable used to determine which partition each message is assigned to. Called (after key serialization): partitioner(key_bytes, all_partitions, available_partitions). The default partitioner implementation hashes each non-None key using the same murmur2 algorithm as the Java client so that messages with the same key are assigned to the same partition. When a key is None, the message is delivered to a random partition (filtered to partitions with available leaders only, if possible).
- **max_request_size** (*int*) – The maximum size of a request. This is also effectively a cap on the maximum record size. Note that the server has its own cap on record size which may be different from this. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests. Default: 1048576.
- **metadata_max_age_ms** (*int*) – The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions. Default: 300000
- **request_timeout_ms** (*int*) – Produce request timeout in milliseconds. As it's sent as part of *ProduceRequest* (it's a blocking call), maximum waiting time can be up to 2 * request_timeout_ms. Default: 40000.
- **retry_backoff_ms** (*int*) – Milliseconds to backoff when retrying on errors. Default: 100.
- **api_version** (*str*) – specify which kafka API version to use. If set to auto, will attempt to infer the broker version by probing various APIs. Default: auto
- **security_protocol** (*str*) – Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL. Default: PLAINTEXT. Default: PLAINTEXT.
- **ssl_context** (*ssl.SSLContext*) – pre-configured SSLContext for wrapping socket connections. Directly passed into asyncio's create_connection(). For more information see *SSL Authentication*. Default: None

- **connections_max_idle_ms** (*int*) – Close idle connections after the number of milliseconds specified by this config. Specifying `None` will disable idle checks. Default: 540000 (9 minutes).
- **enable_idempotence** (*bool*) – When set to `True`, the producer will ensure that exactly one copy of each message is written in the stream. If `False`, producer retries due to broker failures, etc., may write duplicates of the retried message in the stream. Note that enabling idempotence acks to set to `all`. If it is not explicitly set by the user it will be chosen. If incompatible values are set, a `ValueError` will be thrown. New in version 0.5.0.
- **sasl_mechanism** (*str*) – Authentication mechanism when security_protocol is configured for `SASL_PLAINTEXT` or `SASL_SSL`. Valid values are: `PLAIN`, `GSSAPI`, `SCRAM-SHA-256`, `SCRAM-SHA-512`, `OAUTHBEARER`. Default: `PLAIN`
- **sasl_plain_username** (*str*) – username for SASL `PLAIN` authentication. Default: `None`
- **sasl_plain_password** (*str*) – password for SASL `PLAIN` authentication. Default: `None`
- **sasl_oauth_token_provider** (*AbstractTokenProvider*) – OAuthBearer token provider instance. (See `kafka.oauth.abstract`). Default: `None`

Note: Many configuration parameters are taken from the Java client: https://kafka.apache.org/documentation.html#producerconfigs

create_batch()

Create and return an empty `BatchBuilder`.

The batch is not queued for send until submission to `send_batch()`.

Returns empty batch to be filled and submitted by the caller.

Return type *BatchBuilder*

async **flush**()

Wait until all batches are Delivered and futures resolved

async **partitions_for**(*topic*)

Returns set of all known partitions for the topic.

async **send**(*topic*, *value=None*, *key=None*, *partition=None*, *timestamp_ms=None*, *headers=None*)

Publish a message to a topic.

Parameters

- **topic** (*str*) – topic where the message will be published
- **value** (*Optional*) – message value. Must be type `bytes`, or be serializable to `bytes` via configured *value_serializer*. If value is `None`, key is required and message acts as a `delete`.

  See Kafka compaction documentation for more details. (compaction requires kafka >= 0.8.1)
- **partition** (*int, Optional*) – optionally specify a partition. If not set, the partition will be selected using the configured *partitioner*.
- **key** (*Optional*) – a key to associate with the message. Can be used to determine which partition to send the message to. If partition is `None` (and producer's partitioner config is left as default), then messages with the same key will be delivered to the same partition (but if key is `None`, partition is chosen randomly). Must be type `bytes`, or be serializable to bytes via configured *key_serializer*.

- **timestamp_ms** (*int, Optional*) – epoch milliseconds (from Jan 1 1970 UTC) to use as the message timestamp. Defaults to current time.
- **headers** (*Optional*) – Kafka headers to be included in the message using the format `[("key", b"value")]`. Iterable of tuples where key is a normal string and value is a byte string.

Returns object that will be set when message is processed

Return type asyncio.Future

Raises *KafkaTimeoutError* – if we can't schedule this record (pending buffer is full) in up to *request_timeout_ms* milliseconds.

Note: The returned future will wait based on *request_timeout_ms* setting. Cancelling the returned future **will not** stop event from being sent, but cancelling the `send()` coroutine itself **will**.

`async send_and_wait(`*topic*, *value=None*, *key=None*, *partition=None*, *timestamp_ms=None*, *headers=None*`)`

Publish a message to a topic and wait the result

`async send_batch(`*batch*, *topic*, *, *partition*`)`

Submit a BatchBuilder for publication.

Parameters

- **batch** (`BatchBuilder`) – batch object to be published.
- **topic** (*str*) – topic where the batch will be published.
- **partition** (*int*) – partition where this batch will be published.

Returns

object that will be set when the batch is delivered.

Return type asyncio.Future

`async start()`

Connect to Kafka cluster and check server version

`async stop()`

Flush all pending data and close all connections to kafka cluster

`transaction()`

Start a transaction context

4.4.2 Consumer class

class aiokafka.**AIOKafkaConsumer**(**topics*, *loop=None*, *bootstrap_servers='localhost'*, *client_id='aiokafka-0.8.0'*, *group_id=None*, *key_deserializer=None*, *value_deserializer=None*, *fetch_max_wait_ms=500*, *fetch_max_bytes=52428800*, *fetch_min_bytes=1*, *max_partition_fetch_bytes=1048576*, *request_timeout_ms=40000*, *retry_backoff_ms=100*, *auto_offset_reset='latest'*, *enable_auto_commit=True*, *auto_commit_interval_ms=5000*, *check_crcs=True*, *metadata_max_age_ms=300000*, *partition_assignment_strategy=(<class 'kafka.coordinator.assignors.roundrobin.RoundRobinPartitionAssignor'>,)*, *max_poll_interval_ms=300000*, *rebalance_timeout_ms=None*, *session_timeout_ms=10000*, *heartbeat_interval_ms=3000*, *consumer_timeout_ms=200*, *max_poll_records=None*, *ssl_context=None*, *security_protocol='PLAINTEXT'*, *api_version='auto'*, *exclude_internal_topics=True*, *connections_max_idle_ms=540000*, *isolation_level='read_uncommitted'*, *sasl_mechanism='PLAIN'*, *sasl_plain_password=None*, *sasl_plain_username=None*, *sasl_kerberos_service_name='kafka'*, *sasl_kerberos_domain_name=None*, *sasl_oauth_token_provider=None*)

A client that consumes records from a Kafka cluster.

The consumer will transparently handle the failure of servers in the Kafka cluster, and adapt as topic-partitions are created or migrate between brokers.

It also interacts with the assigned Kafka Group Coordinator node to allow multiple consumers to load balance consumption of topics (feature of Kafka >= 0.9.0.0).

Parameters

- ***topics** (*list(str)*) – optional list of topics to subscribe to. If not set, call `subscribe()` or `assign()` before consuming records. Passing topics directly is same as calling `subscribe()` API.
- **bootstrap_servers** (*str, list(str)*) – a `host[:port]` string (or list of `host[:port]` strings) that the consumer should contact to bootstrap initial cluster metadata.

  This does not have to be the full node list. It just needs to have at least one broker that will respond to a Metadata API Request. Default port is 9092. If no servers are specified, will default to `localhost:9092`.
- **client_id** (*str*) – a name for this client. This string is passed in each request to servers and can be used to identify specific server-side log entries that correspond to this client. Also submitted to `GroupCoordinator` for logging with respect to consumer group administration. Default: `aiokafka-{version}`
- **group_id** (*str or None*) – name of the consumer group to join for dynamic partition assignment (if enabled), and to use for fetching and committing offsets. If None, auto-partition assignment (via group coordinator) and offset commits are disabled. Default: None
- **key_deserializer** (*Callable*) – Any callable that takes a raw message key and returns a deserialized key.
- **value_deserializer** (*Callable, Optional*) – Any callable that takes a raw message value and returns a deserialized value.

- **fetch_min_bytes** (*int*) – Minimum amount of data the server should return for a fetch request, otherwise wait up to *fetch_max_wait_ms* for more data to accumulate. Default: 1.
- **fetch_max_bytes** (*int*) – The maximum amount of data the server should return for a fetch request. This is not an absolute maximum, if the first message in the first non-empty partition of the fetch is larger than this value, the message will still be returned to ensure that the consumer can make progress. NOTE: consumer performs fetches to multiple brokers in parallel so memory usage will depend on the number of brokers containing partitions for the topic. Supported Kafka version >= 0.10.1.0. Default: 52428800 (50 Mb).
- **fetch_max_wait_ms** (*int*) – The maximum amount of time in milliseconds the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by fetch_min_bytes. Default: 500.
- **max_partition_fetch_bytes** (*int*) – The maximum amount of data per-partition the server will return. The maximum total memory used for a request = `#partitions * max_partition_fetch_bytes`. This size must be at least as large as the maximum message size the server allows or else it is possible for the producer to send messages larger than the consumer can fetch. If that happens, the consumer can get stuck trying to fetch a large message on a certain partition. Default: 1048576.
- **max_poll_records** (*int*) – The maximum number of records returned in a single call to `getmany()`. Defaults `None`, no limit.
- **request_timeout_ms** (*int*) – Client request timeout in milliseconds. Default: 40000.
- **retry_backoff_ms** (*int*) – Milliseconds to backoff when retrying on errors. Default: 100.
- **auto_offset_reset** (*str*) – A policy for resetting offsets on `OffsetOutOfRangeError` errors: `earliest` will move to the oldest available message, `latest` will move to the most recent, and `none` will raise an exception so you can handle this case. Default: `latest`.
- **enable_auto_commit** (*bool*) – If true the consumer's offset will be periodically committed in the background. Default: True.
- **auto_commit_interval_ms** (*int*) – milliseconds between automatic offset commits, if enable_auto_commit is True. Default: 5000.
- **check_crcs** (*bool*) – Automatically check the CRC32 of the records consumed. This ensures no on-the-wire or on-disk corruption to the messages occurred. This check adds some overhead, so it may be disabled in cases seeking extreme performance. Default: True
- **metadata_max_age_ms** (*int*) – The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions. Default: 300000
- **partition_assignment_strategy** (*list*) – List of objects to use to distribute partition ownership amongst consumer instances when group management is used. This preference is implicit in the order of the strategies in the list. When assignment strategy changes: to support a change to the assignment strategy, new versions must enable support both for the old assignment strategy and the new one. The coordinator will choose the old assignment strategy until all members have been updated. Then it will choose the new strategy. Default: [`RoundRobinPartitionAssignor`]
- **max_poll_interval_ms** (*int*) – Maximum allowed time between calls to consume messages (e.g., `getmany()`). If this interval is exceeded the consumer is considered failed and the group will rebalance in order to reassign the partitions to another consumer group member. If API methods block waiting for messages, that time does not count against this timeout. See KIP-62 for more information. Default 300000

- **rebalance_timeout_ms** (*int*) – The maximum time server will wait for this consumer to rejoin the group in a case of rebalance. In Java client this behaviour is bound to *max.poll.interval.ms* configuration, but as `aiokafka` will rejoin the group in the background, we decouple this setting to allow finer tuning by users that use *ConsumerRebalanceListener* to delay rebalacing. Defaults to `session_timeout_ms`
- **session_timeout_ms** (*int*) – Client group session and failure detection timeout. The consumer sends periodic heartbeats (*heartbeat.interval.ms*) to indicate its liveness to the broker. If no hearts are received by the broker for a group member within the session timeout, the broker will remove the consumer from the group and trigger a rebalance. The allowed range is configured with the **broker** configuration properties *group.min.session.timeout.ms* and *group.max.session.timeout.ms*. Default: 10000
- **heartbeat_interval_ms** (*int*) – The expected time in milliseconds between heartbeats to the consumer coordinator when using Kafka's group management feature. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than *session_timeout_ms*, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances. Default: 3000
- **consumer_timeout_ms** (*int*) – maximum wait timeout for background fetching routine. Mostly defines how fast the system will see rebalance and request new data for new partitions. Default: 200
- **api_version** (*str*) – specify which kafka API version to use. *AIOKafkaConsumer* supports Kafka API versions >=0.9 only. If set to `auto`, will attempt to infer the broker version by probing various APIs. Default: `auto`
- **security_protocol** (*str*) – Protocol used to communicate with brokers. Valid values are: `PLAINTEXT`, `SSL`. Default: `PLAINTEXT`.
- **ssl_context** (*ssl.SSLContext*) – pre-configured `SSLContext` for wrapping socket connections. Directly passed into asyncio's `create_connection()`. For more information see *SSL Authentication*. Default: None.
- **exclude_internal_topics** (*bool*) – Whether records from internal topics (such as offsets) should be exposed to the consumer. If set to True the only way to receive records from an internal topic is subscribing to it. Requires 0.10+ Default: True
- **connections_max_idle_ms** (*int*) – Close idle connections after the number of milliseconds specified by this config. Specifying *None* will disable idle checks. Default: 540000 (9 minutes).
- **isolation_level** (*str*) – Controls how to read messages written transactionally.

  If set to `read_committed`, *getmany()* will only return transactional messages which have been committed. If set to `read_uncommitted` (the default), *getmany()* will return all messages, even transactional messages which have been aborted.

  Non-transactional messages will be returned unconditionally in either mode.

  Messages will always be returned in offset order. Hence, in *read_committed* mode, *getmany()* will only return messages up to the last stable offset (LSO), which is the one less than the offset of the first open transaction. In particular any messages appearing after messages belonging to ongoing transactions will be withheld until the relevant transaction has been completed. As a result, *read_committed* consumers will not be able to read up to the high watermark when there are in flight transactions. Further, when in *read_committed* the seek_to_end method will return the LSO. See method docs below. Default: `read_uncommitted`

- **sasl_mechanism** (*str*) – Authentication mechanism when security_protocol is configured for `SASL_PLAINTEXT` or `SASL_SSL`. Valid values are: `PLAIN`, `GSSAPI`, `SCRAM-SHA-256`, `SCRAM-SHA-512`, `OAUTHBEARER`. Default: `PLAIN`
- **sasl_plain_username** (*str*) – username for SASL `PLAIN` authentication. Default: None
- **sasl_plain_password** (*str*) – password for SASL `PLAIN` authentication. Default: None
- **sasl_oauth_token_provider** (`AbstractTokenProvider`) – OAuthBearer token provider instance. (See `kafka.oauth.abstract`). Default: None

Note: Many configuration parameters are taken from Java Client: https://kafka.apache.org/documentation.html#newconsumerconfigs

`assign`(*partitions*)

Manually assign a list of `TopicPartition` to this consumer.

This interface does not support incremental assignment and will replace the previous assignment (if there was one).

Parameters **partitions** (*list(*`TopicPartition`*)*) – assignment for this instance.

Raises **IllegalStateError** – if consumer has already called `subscribe()`

Warning: It is not possible to use both manual partition assignment with `assign()` and group assignment with `subscribe()`.

Note: Manual topic assignment through this method does not use the consumer's group management functionality. As such, there will be **no rebalance operation triggered** when group membership or cluster and topic metadata change.

`assignment`()

Get the set of partitions currently assigned to this consumer.

If partitions were directly assigned using `assign()`, then this will simply return the same partitions that were previously assigned.

If topics were subscribed using `subscribe()`, then this will give the set of topic partitions currently assigned to the consumer (which may be empty if the assignment hasn't happened yet or if the partitions are in the process of being reassigned).

Returns the set of partitions currently assigned to this consumer

Return type set(*TopicPartition*)

`async beginning_offsets`(*partitions*)

Get the first offset for the given partitions.

This method does not change the current consumer position of the partitions.

Note: This method may block indefinitely if the partition does not exist.

Parameters **partitions** (*list[*`TopicPartition`*]*) – List of `TopicPartition` instances to fetch offsets for.

Returns mapping of partition to earliest available offset.

Return type dict [*TopicPartition*, int]

Raises

- ***UnsupportedVersionError*** – If the broker does not support looking up the offsets by timestamp.
- ***KafkaTimeoutError*** – If fetch failed in *request_timeout_ms*.

New in version 0.3.0.

async **commit**(*offsets=None*)

Commit offsets to Kafka.

This commits offsets only to Kafka. The offsets committed using this API will be used on the first fetch after every rebalance and also on startup. As such, if you need to store offsets in anything other than Kafka, this API should not be used.

Currently only supports kafka-topic offset storage (not Zookeeper)

When explicitly passing *offsets* use either offset of next record, or tuple of offset and metadata:

```
tp = TopicPartition(msg.topic, msg.partition)
metadata = "Some utf-8 metadata"
# Either
await consumer.commit({tp: msg.offset + 1})
# Or position directly
await consumer.commit({tp: (msg.offset + 1, metadata)})
```

Note: If you want *fire and forget* commit, like `commit_async()` in kafka-python, just run it in a task. Something like:

```
fut = loop.create_task(consumer.commit())
fut.add_done_callback(on_commit_done)
```

Parameters **offsets** (*dict, Optional*) – A mapping from *TopicPartition* to `(offset, metadata)` to commit with the configured `group_id`. Defaults to current consumed offsets for all subscribed partitions.

Raises

- ***CommitFailedError*** – If membership already changed on broker.
- ***IllegalOperation*** – If used with `group_id == None`.
- ***IllegalStateError*** – If partitions not assigned.
- ***KafkaError*** – If commit failed on broker side. This could be due to invalid offset, too long metadata, authorization failure, etc.
- **ValueError** – If offsets is of wrong format.

Changed in version 0.4.0: Changed `AssertionError` to *IllegalStateError* in case of unassigned partition.

Changed in version 0.4.0: Will now raise *CommitFailedError* in case membership changed, as (possibly) this partition is handled by another consumer.

async **committed**(*partition*)

Get the last committed offset for the given partition. (whether the commit happened by this process or another).

This offset will be used as the position for the consumer in the event of a failure.

This call will block to do a remote call to get the latest offset, as those are not cached by consumer (Transactional Producer can change them without Consumer knowledge as of Kafka 0.11.0)

Parameters **partition** (`TopicPartition`) – the partition to check

Returns The last committed offset, or None if there was no prior commit.

Raises *IllegalOperation* – If used with `group_id == None`

async **end_offsets**(*partitions*)

Get the last offset for the given partitions. The last offset of a partition is the offset of the upcoming message, i.e. the offset of the last available message + 1.

This method does not change the current consumer position of the partitions.

Note: This method may block indefinitely if the partition does not exist.

Parameters **partitions** (*list[TopicPartition]*) – List of *TopicPartition* instances to fetch offsets for.

Returns mapping of partition to last available offset + 1.

Return type dict [*TopicPartition*, int]

Raises

- *UnsupportedVersionError* – If the broker does not support looking up the offsets by timestamp.
- *KafkaTimeoutError* – If fetch failed in `request_timeout_ms`

New in version 0.3.0.

async **getmany**(**partitions*, *timeout_ms=0*, *max_records=None*)

Get messages from assigned topics / partitions.

Prefetched messages are returned in batches by topic-partition. If messages is not available in the prefetched buffer this method waits *timeout_ms* milliseconds.

Parameters

- **partitions** (*list[TopicPartition]*) – The partitions that need fetching message. If no one partition specified then all subscribed partitions will be used
- **timeout_ms** (*int, Optional*) – milliseconds spent waiting if data is not available in the buffer. If 0, returns immediately with any records that are available currently in the buffer, else returns empty. Must not be negative. Default: 0

Returns topic to list of records since the last fetch for the subscribed list of topics and partitions

Return type dict(*TopicPartition*, list[*ConsumerRecord*])

Example usage:

```
data = await consumer.getmany()
for tp, messages in data.items():
    topic = tp.topic
    partition = tp.partition
    for message in messages:
        # Process message
        print(message.offset, message.key, message.value)
```

async getone(**partitions*)

Get one message from Kafka. If no new messages prefetched, this method will wait for it.

Parameters **partitions** (*list*(`TopicPartition`)) – Optional list of partitions to return from. If no partitions specified then returned message will be from any partition, which consumer is subscribed to.

Returns the message

Return type *ConsumerRecord*

Will return instance of

```
collections.namedtuple(
    "ConsumerRecord",
    ["topic", "partition", "offset", "key", "value"])
```

Example usage:

```
while True:
    message = await consumer.getone()
    topic = message.topic
    partition = message.partition
    # Process message
    print(message.offset, message.key, message.value)
```

highwater(*partition*)

Last known highwater offset for a partition.

A highwater offset is the offset that will be assigned to the next message that is produced. It may be useful for calculating lag, by comparing with the reported position. Note that both position and highwater refer to the *next* offset – i.e., highwater offset is one greater than the newest available message.

Highwater offsets are returned as part of `FetchResponse`, so will not be available if messages for this partition were not requested yet.

Parameters **partition** (`TopicPartition`) – partition to check

Returns offset if available

Return type int or None

last_poll_timestamp(*partition*)

Returns the timestamp of the last poll of this partition (in ms). It is the last time `highwater()` and `last_stable_offset()` were updated. However it does not mean that new messages were received.

As with `highwater()` will not be available until some messages are consumed.

Parameters **partition** (`TopicPartition`) – partition to check

Returns timestamp if available

Return type int or None

`last_stable_offset`(*partition*)

Returns the Last Stable Offset of a topic. It will be the last offset up to which point all transactions were completed. Only available in with isolation_level *read_committed*, in *read_uncommitted* will always return -1. Will return None for older Brokers.

As with `highwater()` will not be available until some messages are consumed.

Parameters **partition** (`TopicPartition`) – partition to check

Returns offset if available

Return type int or None

`async offsets_for_times`(*timestamps*)

Look up the offsets for the given partitions by timestamp. The returned offset for each partition is the earliest offset whose timestamp is greater than or equal to the given timestamp in the corresponding partition.

The consumer does not have to be assigned the partitions.

If the message format version in a partition is before 0.10.0, i.e. the messages do not have timestamps, `None` will be returned for that partition.

Note: This method may block indefinitely if the partition does not exist.

Parameters **timestamps** (`dict(TopicPartition, int)`) – mapping from partition to the timestamp to look up. Unit should be milliseconds since beginning of the epoch (midnight Jan 1, 1970 (UTC))

Returns mapping from partition to the timestamp and offset of the first message with timestamp greater than or equal to the target timestamp.

Return type dict(*TopicPartition*, *OffsetAndTimestamp*)

Raises

- `ValueError` – If the target timestamp is negative
- `UnsupportedVersionError` – If the broker does not support looking up the offsets by timestamp.
- `KafkaTimeoutError` – If fetch failed in *request_timeout_ms*

New in version 0.3.0.

`partitions_for_topic`(*topic*)

Get metadata about the partitions for a given topic.

This method will return *None* if Consumer does not already have metadata for this topic.

Parameters **topic** (`str`) – topic to check

Returns partition ids

Return type set

`pause`(**partitions*)

Suspend fetching from the requested partitions.

Future calls to `getmany()` will not return any records from these partitions until they have been resumed using `resume()`.

Note: This method does not affect partition subscription. In particular, it does not cause a group rebalance when automatic assignment is used.

Parameters ***partitions** (*list[TopicPartition]*) – Partitions to pause.

paused()

Get the partitions that were previously paused using *pause()*.

Returns partitions

Return type set[*TopicPartition*]

async position(*partition*)

Get the offset of the *next record* that will be fetched (if a record with that offset exists on broker).

Parameters **partition** (TopicPartition) – partition to check

Returns offset

Return type int

Raises ***IllegalStateError*** – partition is not assigned

Changed in version 0.4.0: Changed AssertionError to *IllegalStateError* in case of unassigned partition

resume(**partitions*)

Resume fetching from the specified (paused) partitions.

Parameters ***partitions** (*list[TopicPartition]*) – Partitions to resume.

seek(*partition*, *offset*)

Manually specify the fetch offset for a *TopicPartition*.

Overrides the fetch offsets that the consumer will use on the next *getmany()*/*getone()* call. If this API is invoked for the same partition more than once, the latest offset will be used on the next fetch.

Note: You may lose data if this API is arbitrarily used in the middle of consumption to reset the fetch offsets. Use it either on rebalance listeners or after all pending messages are processed.

Parameters

- **partition** (TopicPartition) – partition for seek operation
- **offset** (*int*) – message offset in partition

Raises

- **ValueError** – if offset is not a positive integer
- ***IllegalStateError*** – partition is not currently assigned

Changed in version 0.4.0: Changed AssertionError to *IllegalStateError* and ValueError in respective cases.

async seek_to_beginning(**partitions*)

Seek to the oldest available offset for partitions.

Parameters ***partitions** – Optionally provide specific *TopicPartition*, otherwise default to all assigned partitions.

Raises

- ***IllegalStateError*** – If any partition is not currently assigned
- **TypeError** – If partitions are not instances of *TopicPartition*

New in version 0.3.0.

async seek_to_committed(**partitions*)

Seek to the committed offset for partitions.

Parameters ***partitions** – Optionally provide specific *TopicPartition*, otherwise default to all assigned partitions.

Returns mapping of the currently committed offsets.

Return type dict(*TopicPartition*, int)

Raises

- ***IllegalStateError*** – If any partition is not currently assigned
- ***IllegalOperation*** – If used with `group_id == None`

Changed in version 0.3.0: Changed `AssertionError` to *IllegalStateError* in case of unassigned partition

async seek_to_end(**partitions*)

Seek to the most recent available offset for partitions.

Parameters ***partitions** – Optionally provide specific *TopicPartition*, otherwise default to all assigned partitions.

Raises

- ***IllegalStateError*** – If any partition is not currently assigned
- **TypeError** – If partitions are not instances of *TopicPartition*

New in version 0.3.0.

async start()

Connect to Kafka cluster. This will:

- Load metadata for all cluster nodes and partition allocation
- Wait for possible topic autocreation
- Join group if `group_id` provided

async stop()

Close the consumer, while waiting for finalizers:

- Commit last consumed message if autocommit enabled
- Leave group if used Consumer Groups

subscribe(*topics=()*, *pattern=None*, *listener=None*)

Subscribe to a list of topics, or a topic regex pattern.

Partitions will be dynamically assigned via a group coordinator. Topic subscriptions are not incremental: this list will replace the current assignment (if there is one).

This method is incompatible with *assign()*.

Parameters

- **topics** (*list*) – List of topics for subscription.
- **pattern** (*str*) – Pattern to match available topics. You must provide either topics or pattern, but not both.

- **listener** (ConsumerRebalanceListener) – Optionally include listener callback, which will be called before and after each rebalance operation. As part of group management, the consumer will keep track of the list of consumers that belong to a particular group and will trigger a rebalance operation if one of the following events trigger:
  - Number of partitions change for any of the subscribed topics
  - Topic is created or deleted
  - An existing member of the consumer group dies
  - A new member is added to the consumer group

  When any of these events are triggered, the provided listener will be invoked first to indicate that the consumer's assignment has been revoked, and then again when the new assignment has been received. Note that this listener will immediately override any listener set in a previous call to subscribe. It is guaranteed, however, that the partitions revoked/assigned through this interface are from topics subscribed in this call.

Raises

- *IllegalStateError* – if called after previously calling *assign()*
- ValueError – if neither topics or pattern is provided or both are provided
- TypeError – if listener is not a *ConsumerRebalanceListener*

subscription()

Get the current topics subscription.

Returns a set of topics

Return type frozenset(str)

async topics()

Get all topics the user is authorized to view.

Returns topics

Return type set

unsubscribe()

Unsubscribe from all topics and clear all assigned partitions.

4.4.3 Helpers

aiokafka.helpers.**create_ssl_context**(*, *cafile=None*, *capath=None*, *cadata=None*, *certfile=None*, *keyfile=None*, *password=None*, *crlfile=None*)

Simple helper, that creates an SSLContext based on params similar to those in kafka-python, but with some restrictions like:

- *check_hostname* is not optional, and will be set to True
- *crlfile* option is missing. It is fairly hard to test it.

Parameters

- **cafile** (*str*) – Certificate Authority file path containing certificates used to sign broker certificates. If CA not specified (by either cafile, capath, cadata) default system CA will be used if found by OpenSSL. For more information see load_verify_locations(). Default: None

- **capath** (*str*) – Same as *cafile*, but points to a directory containing several CA certificates. For more information see `load_verify_locations()`. Default: `None`
- **cadata** (*str, bytes*) – Same as *cafile*, but instead contains already read data in either ASCII or bytes format. Can be used to specify DER-encoded certificates, rather than PEM ones. For more information see `load_verify_locations()`. Default: `None`
- **certfile** (*str*) – optional filename of file in PEM format containing the client certificate, as well as any CA certificates needed to establish the certificate's authenticity. For more information see `load_cert_chain()`. Default: `None`.
- **keyfile** (*str*) – optional filename containing the client private key. For more information see `load_cert_chain()`. Default: `None`.
- **password** (*str*) – optional password to be used when loading the certificate chain. For more information see `load_cert_chain()`. Default: `None`.

4.4.4 Abstracts

class aiokafka.abc.AbstractTokenProvider(***config*)

A Token Provider must be used for the SASL OAuthBearer protocol.

The implementation should ensure token reuse so that multiple calls at connect time do not create multiple tokens. The implementation should also periodically refresh the token in order to guarantee that each call returns an unexpired token.

A timeout error should be returned after a short period of inactivity so that the broker can log debugging info and retry.

Token Providers MUST implement the `token()` method

abstract async token()

An async callback returning a `str` ID/Access Token to be sent to the Kafka client. In case where a synchoronous callback is needed, implementations like following can be used:

```
from aiokafka.abc import AbstractTokenProvider

class CustomTokenProvider(AbstractTokenProvider):
    async def token(self):
        return asyncio.get_running_loop().run_in_executor(
            None, self._token)

    def _token(self):
        # The actual synchoronous token callback.
```

extensions()

This is an OPTIONAL method that may be implemented.

Returns a map of key-value pairs that can be sent with the SASL/OAUTHBEARER initial client request. If not implemented, the values are ignored

This feature is only available in Kafka >= 2.1.0.

class aiokafka.abc.ConsumerRebalanceListener

A callback interface that the user can implement to trigger custom actions when the set of partitions assigned to the consumer changes.

This is applicable when the consumer is having Kafka auto-manage group membership. If the consumer's directly assign partitions, those partitions will never be reassigned and this callback is not applicable.

When Kafka is managing the group membership, a partition re-assignment will be triggered any time the members of the group changes or the subscription of the members changes. This can occur when processes die, new process instances are added or old instances come back to life after failure. Rebalances can also be triggered by changes affecting the subscribed topics (e.g. when then number of partitions is administratively adjusted).

There are many uses for this functionality. One common use is saving offsets in a custom store. By saving offsets in the `on_partitions_revoked()`, call we can ensure that any time partition assignment changes the offset gets saved.

Another use is flushing out any kind of cache of intermediate results the consumer may be keeping. For example, consider a case where the consumer is subscribed to a topic containing user page views, and the goal is to count the number of page views per users for each five minute window. Let's say the topic is partitioned by the user id so that all events for a particular user will go to a single consumer instance. The consumer can keep in memory a running tally of actions per user and only flush these out to a remote data store when its cache gets too big. However if a partition is reassigned it may want to automatically trigger a flush of this cache, before the new owner takes over consumption.

This callback will execute during the rebalance process, and Consumer will wait for callbacks to finish before proceeding with group join.

It is guaranteed that all consumer processes will invoke `on_partitions_revoked()` prior to any process invoking `on_partitions_assigned()`. So if offsets or other state is saved in the `on_partitions_revoked()` call, it should be saved by the time the process taking over that partition has their `on_partitions_assigned()` callback called to load the state.

abstract on_partitions_revoked(*revoked*)

A coroutine or function the user can implement to provide cleanup or custom state save on the start of a rebalance operation.

This method will be called *before* a rebalance operation starts and *after* the consumer stops fetching data.

If you are using manual commit you have to commit all consumed offsets here, to avoid duplicate message delivery after rebalance is finished.

Note: This method is only called before rebalances. It is not called prior to `AIOKafkaConsumer.stop()`

Parameters **revoked** (`list(TopicPartition)`) – the partitions that were assigned to the consumer on the last rebalance

abstract on_partitions_assigned(*assigned*)

A coroutine or function the user can implement to provide load of custom consumer state or cache warmup on completion of a successful partition re-assignment.

This method will be called *after* partition re-assignment completes and *before* the consumer starts fetching data again.

It is guaranteed that all the processes in a consumer group will execute their `on_partitions_revoked()` callback before any instance executes its `on_partitions_assigned()` callback.

Parameters **assigned** (`list(TopicPartition)`) – the partitions assigned to the consumer (may include partitions that were previously assigned)

4.4.5 SSL Authentication

Security is not an easy thing, at least when you want to do it right. Before diving in on how to setup *aiokafka* to work with SSL, make sure there is a need for SSL Authentication and go through the official documentation for SSL support in Kafka itself.

aiokafka provides only `ssl_context` as a parameter for Consumer and Producer classes. This is done intentionally, as it is recommended that you read through the Python ssl documentation to have some understanding on the topic. Although if you know what you are doing, there is a simple helper function *aiokafka.helpers.create_ssl_context()*, that will create an `ssl.SSLContext` based on similar params to kafka-python.

A few notes on Kafka's SSL store types. Java uses **JKS** store type, that contains normal certificates, same as ones OpenSSL (and Python, as it's based on OpenSSL) uses, but encodes them into a single, encrypted file, protected by another password. Just look the internet on how to extract *CARoot*, *Certificate* and *Key* from JKS store.

See also the *Using SSL with aiokafka* example.

4.4.6 SASL Authentication

As of version 0.5.1 aiokafka supports SASL authentication using both `PLAIN` and `GSSAPI` SASL methods. Be sure to install gssapi python module to use `GSSAPI`.

Please consult the official documentation for setup instructions on Broker side. Client configuration is pretty much the same as Java's, consult the `sasl_*` options in Consumer and Producer API Reference for more details.

4.4.7 Error handling

Both consumer and producer can raise exceptions that inherit from the *aiokafka.errors.KafkaError* class.

Exception handling example:

```
from aiokafka.errors import KafkaError, KafkaTimeoutError
# ...
try:
    send_future = await producer.send('foobar', b'test data')
    response = await send_future  #  wait until message is produced
except KafkaTimeoutError:
    print("produce timeout... maybe we want to resend data again?")
except KafkaError as err:
    print("some kafka error on produce: {}".format(err))
```

Consumer errors

Consumer's `async for` and *getone()*/*getmany()* interfaces will handle those differently. Possible consumer errors include:

- *TopicAuthorizationFailedError* - topic requires authorization. Always raised
- *OffsetOutOfRangeError* - if you don't specify *auto_offset_reset* policy and started cosumption from not valid offset. Always raised
- *RecordTooLargeError* - broker has a *MessageSet* larger than *max_partition_fetch_bytes*. **async for** - log error, **get*** will raise it.
- *InvalidMessageError* - CRC check on MessageSet failed due to connection failure or bug. Always raised. Changed in version `0.5.0`, before we ignored this error in `async for`.

4.4.8 Other references

class aiokafka.producer.message_accumulator.**BatchBuilder**(*magic*, *batch_size*, *compression_type*, *, *is_transactional*)

class aiokafka.consumer.group_coordinator.**GroupCoordinator**(*client*, *subscription*, *, *group_id='aiokafka-default-group'*, *session_timeout_ms=10000*, *heartbeat_interval_ms=3000*, *retry_backoff_ms=100*, *enable_auto_commit=True*, *auto_commit_interval_ms=5000*, *assignors=(<class 'kafka.coordinator.assignors.roundrobin.RoundRobinParti* *), exclude_internal_topics=True*, *max_poll_interval_ms=300000*, *rebalance_timeout_ms=30000*)

GroupCoordinator implements group management for single group member by interacting with a designated Kafka broker (the coordinator). Group semantics are provided by extending this class.

From a high level, Kafka's group management protocol consists of the following sequence of actions:

1. Group Registration: Group members register with the coordinator providing their own metadata (such as the set of topics they are interested in).
2. Group/Leader Selection: The coordinator (one of Kafka nodes) select the members of the group and chooses one member (one of client's) as the leader.
3. State Assignment: The leader receives metadata for all members and assigns partitions to them.
4. Group Stabilization: Each member receives the state assigned by the leader and begins processing. Between each phase coordinator awaits all clients to respond. If some do not respond in time - it will revoke their membership

NOTE: Try to maintain same log messages and behaviour as Java and

kafka-python clients:

https://github.com/apache/kafka/blob/0.10.1.1/clients/src/main/java/ org/apache/kafka/clients/consumer/internals/AbstractC
https://github.com/apache/kafka/blob/0.10.1.1/clients/src/main/java/ org/apache/kafka/clients/consumer/internals/Consume

class kafka.coordinator.assignors.roundrobin.**RoundRobinPartitionAssignor**

The roundrobin assignor lays out all the available partitions and all the available consumers. It then proceeds to do a roundrobin assignment from partition to consumer. If the subscriptions of all consumer instances are identical, then the partitions will be uniformly distributed. (i.e., the partition ownership counts will be within a delta of exactly one across all consumers.)

For example, suppose there are two consumers C0 and C1, two topics t0 and t1, and each topic has 3 partitions, resulting in partitions t0p0, t0p1, t0p2, t1p0, t1p1, and t1p2.

The assignment will be: C0: [t0p0, t0p2, t1p1] C1: [t0p1, t1p0, t1p2]

When subscriptions differ across consumer instances, the assignment process still considers each consumer instance in round robin fashion but skips over an instance if it is not subscribed to the topic. Unlike the case when subscriptions are identical, this can result in imbalanced assignments.

For example, suppose we have three consumers C0, C1, C2, and three topics t0, t1, t2, with unbalanced partitions t0p0, t1p0, t1p1, t2p0, t2p1, t2p2, where C0 is subscribed to t0; C1 is subscribed to t0, t1; and C2 is subscribed to t0, t1, t2.

The assignment will be: C0: [t0p0] C1: [t1p0] C2: [t1p1, t2p0, t2p1, t2p2]

Errors

exception aiokafka.errors.**ConcurrentTransactions**

exception aiokafka.errors.**ConsumerStoppedError**
Raised on *get** methods of Consumer if it's cancelled, even pending ones.

exception aiokafka.errors.**CoordinatorLoadInProgressError**

exception aiokafka.errors.**CoordinatorNotAvailableError**

exception aiokafka.errors.**DelegationTokenAuthDisabled**

exception aiokafka.errors.**DelegationTokenAuthorizationFailed**

exception aiokafka.errors.**DelegationTokenExpired**

exception aiokafka.errors.**DelegationTokenNotFound**

exception aiokafka.errors.**DelegationTokenOwnerMismatch**

exception aiokafka.errors.**DelegationTokenRequestNotAllowed**

exception aiokafka.errors.**DuplicateSequenceNumber**

exception aiokafka.errors.**FetchSessionIdNotFound**

aiokafka.errors.**GroupCoordinatorNotAvailableError**
alias of *aiokafka.errors.CoordinatorNotAvailableError*

exception aiokafka.errors.**GroupIdNotFound**

aiokafka.errors.**GroupLoadInProgressError**
alias of *aiokafka.errors.CoordinatorLoadInProgressError*

exception aiokafka.errors.**IllegalOperation**
Raised if you try to execute an operation, that is not available with current configuration. For example trying to commit if no group_id was given.

exception aiokafka.errors.**InvalidFetchSessionEpoch**

exception aiokafka.errors.**InvalidPrincipalType**

exception aiokafka.errors.**InvalidProducerEpoch**

exception aiokafka.errors.**InvalidProducerIdMapping**

exception aiokafka.errors.**InvalidTransactionTimeout**

exception aiokafka.errors.**InvalidTxnState**

exception aiokafka.errors.**KafkaStorageError**

exception aiokafka.errors.**ListenerNotFound**

exception aiokafka.errors.**LogDirNotFound**

exception aiokafka.errors.**NoOffsetForPartitionError**

exception aiokafka.errors.**NonEmptyGroup**

exception aiokafka.errors.**NotCoordinatorError**

aiokafka.errors.**NotCoordinatorForGroupError**
alias of *aiokafka.errors.NotCoordinatorError*

exception aiokafka.errors.**OperationNotAttempted**

exception aiokafka.errors.**OutOfOrderSequenceNumber**

exception aiokafka.errors.**ProducerClosed**

exception aiokafka.errors.**ProducerFenced**(*msg='There is a newer producer using the same transactional_id ortransaction timeout occurred (check that processing time is below transaction_timeout_ms)'*)

Another producer with the same transactional ID went online. NOTE: As it seems this will be raised by Broker if transaction timeout occurred also.

exception aiokafka.errors.**ReassignmentInProgress**

exception aiokafka.errors.**RecordTooLargeError**

exception aiokafka.errors.**SaslAuthenticationFailed**

exception aiokafka.errors.**SecurityDisabled**

exception aiokafka.errors.**TransactionCoordinatorFenced**

exception aiokafka.errors.**TransactionalIdAuthorizationFailed**

exception aiokafka.errors.**UnknownProducerId**

class aiokafka.errors.**KafkaTimeoutError**

class aiokafka.errors.**RequestTimedOutError**

class aiokafka.errors.**NotEnoughReplicasError**

class aiokafka.errors.**NotEnoughReplicasAfterAppendError**

class aiokafka.errors.**KafkaError**

class aiokafka.errors.**UnsupportedVersionError**

class aiokafka.errors.**TopicAuthorizationFailedError**

class aiokafka.errors.**OffsetOutOfRangeError**

class aiokafka.errors.**CorruptRecordException**

class kafka.errors.**CorruptRecordException**

aiokafka.errors.**InvalidMessageError**

alias of *kafka.errors.CorruptRecordException*

class aiokafka.errors.**IllegalStateError**

class aiokafka.errors.**CommitFailedError**(**args*, ***kwargs*)

Structs

class kafka.structs.**TopicPartition**(*topic*, *partition*)

A Kafka broker metadata used by admin tools.

Keyword Arguments

- **nodeID** (*int*) – The Kafka broker id.
- **host** (*str*) – The Kafka broker hostname.
- **port** (*int*) – The Kafka broker port.

- **rack** (*str*) – The rack of the broker, which is used to in rack aware partition assignment for fault tolerance.
- **Examples** – *RACK1*, *us-east-1d*. Default: None

property partition
Alias for field number 1

property topic
Alias for field number 0

class aiokafka.structs.**RecordMetadata**(*topic: str, partition: int, topic_partition: kafka.structs.TopicPartition, offset: int, timestamp: Optional[int], timestamp_type: int, log_start_offset: Optional[int]*)

Returned when a *AIOKafkaProducer* sends a message

property log_start_offset

property offset
The unique offset of the message in this partition.

See *Offsets and Consumer Position* for more details on offsets.

property partition
The partition number

property timestamp
Timestamp in millis, None for older Brokers

property timestamp_type
The timestamp type of this record.

If the broker respected the timestamp passed to *AIOKafkaProducer.send()*, `0` will be returned (`CreateTime`).

If the broker set it's own timestamp, 1 will be returned (`LogAppendTime`).

property topic
The topic name

property topic_partition

class aiokafka.structs.**ConsumerRecord**(**args, **kwds*)

checksum: int
Deprecated

headers: Sequence[Tuple[str, bytes]]
The headers

key: Optional[*aiokafka.structs.KT*]
The key (or *None* if no key is specified)

offset: int
The position of this record in the corresponding Kafka partition.

partition: int
The partition from which this record is received

serialized_key_size: int
The size of the serialized, uncompressed key in bytes.

serialized_value_size: int
The size of the serialized, uncompressed value in bytes.

timestamp: int
The timestamp of this record

timestamp_type: int
The timestamp type of this record

topic: str
The topic this record is received from

value: Optional[*aiokafka.structs.VT*]
The value

class aiokafka.structs.**OffsetAndTimestamp**(*offset*, *timestamp*)

property offset
Alias for field number 0

property timestamp
Alias for field number 1

class aiokafka.structs.**KT**
The type of a key.

class aiokafka.structs.**VT**
The type of a value.

Protocols

kafka.protocol.produce.**ProduceRequest**
alias of [<class 'kafka.protocol.produce.ProduceRequest_v0'>, <class 'kafka.protocol.produce.ProduceRequest_v1'>, <class 'kafka.protocol.produce.ProduceRequest_v2'>, <class 'kafka.protocol.produce.ProduceRequest_v3'>, <class 'kafka.protocol.produce.ProduceRequest_v4'>, <class 'kafka.protocol.produce.ProduceRequest_v5'>, <class 'kafka.protocol.produce.ProduceRequest_v6'>, <class 'kafka.protocol.produce.ProduceRequest_v7'>, <class 'kafka.protocol.produce.ProduceRequest_v8'>]

4.5 Examples

4.5.1 Serialization and compression

Kafka supports several compression types: gzip, snappy and lz4. You only need to specify the compression in Kafka Producer, Consumer will decompress automatically.

Note: Messages are compressed in batches, so you will have more efficiency on larger batches. You can consider setting *linger_ms* to batch more data before sending.

By default *value* and *key* attributes of returned *ConsumerRecord* instances are bytes. You can use custom serializer/deserializer hooks to operate on objects instead of bytes in those attributes.

Producer

```
import json
import asyncio
from aiokafka import AIOKafkaProducer

def serializer(value):
```

(continues on next page)

(continued from previous page)

```
    return json.dumps(value).encode()

async def produce():
    producer = AIOKafkaProducer(
        bootstrap_servers='localhost:9092',
        value_serializer=serializer,
        compression_type="gzip")

    await producer.start()
    data = {"a": 123.4, "b": "some string"}
    await producer.send('foobar', data)
    data = [1,2,3,4]
    await producer.send('foobar', data)
    await producer.stop()

asyncio.run(produce())
```

Consumer

```
import json
import asyncio
from kafka.common import KafkaError
from aiokafka import AIOKafkaConsumer

def deserializer(serialized):
    return json.loads(serialized)

async def consume():
    # consumer will decompress messages automatically
    # in accordance to compression type specified in producer
    consumer = AIOKafkaConsumer(
        'foobar',
        bootstrap_servers='localhost:9092',
        value_deserializer=deserializer,
        auto_offset_reset='earliest')
    await consumer.start()
    data = await consumer.getmany(timeout_ms=10000)
    for tp, messages in data.items():
        for message in messages:
            print(type(message.value), message.value)
    await consumer.stop()

asyncio.run(consume())
```

Output:

```
>>> python3 producer.py
>>> python3 consumer.py
<class 'dict'> {'a': 123.4, 'b': 'some string'}
<class 'list'> [1,2,3,4]
```

4.5.2 Manual commit

When processing more sensitive data `enable_auto_commit=False` mode of Consumer can lead to data loss in cases of critical failure. To avoid it we can commit offsets manually after they were processed. Note, that this is a tradeoff from *at most once* to *at least once* delivery, to achieve *exactly once* you will need to save offsets in the destination database and validate those yourself.

More on message delivery: https://kafka.apache.org/documentation.html#semantics

Note: After Kafka Broker version 0.11 and after *aiokafka==0.5.0* it is possible to use Transactional Producer to achieve *exactly once* delivery semantics. See *Transactional producer* section.

Consumer:

```
import json
import asyncio
from kafka.common import KafkaError
from aiokafka import AIOKafkaConsumer

async def consume():
    consumer = AIOKafkaConsumer(
        'foobar',
        bootstrap_servers='localhost:9092',
        auto_offset_reset='earliest',
        group_id="some-consumer-group",
        enable_auto_commit=False)
    await consumer.start()
    # we want to consume 10 messages from "foobar" topic
    # and commit after that
    for _ in range(10):
        msg = await consumer.getone()
    await consumer.commit()

    await consumer.stop()

asyncio.run(consume())
```

4.5.3 Group consumer

As of Kafka 9.0 Consumers can consume on the same topic simultaneously. This is achieved by coordinating consumers by one of Kafka broker nodes (coordinator). This node will perform synchronization of partition assignment (thou the partitions will be assigned by python code) and consumers will always return messages for the assigned partitions.

Note: Though Consumer will never return messages from not assigned partitions, if you are in `autocommit=False` mode, you should re-check assignment before processing next message returned by `getmany()` call.

Producer:

```
import sys
import asyncio
from aiokafka import AIOKafkaProducer

async def produce(value, partition):
```

(continues on next page)

(continued from previous page)

```
    producer = AIOKafkaProducer(bootstrap_servers='localhost:9092')

    await producer.start()
    await producer.send('some-topic', value, partition=partition)
    await producer.stop()

if len(sys.argv) != 3:
    print("usage: producer.py <partition> <message>")
    sys.exit(1)
value = sys.argv[2].encode()
partition = int(sys.argv[1])

asyncio.run(produce(value, partition))
```

Consumer:

```
import sys
import asyncio
from aiokafka import AIOKafkaConsumer

async def consume():
    consumer = AIOKafkaConsumer(
        'some-topic',
        group_id=group_id,
        bootstrap_servers='localhost:9092',
        auto_offset_reset='earliest')
    await consumer.start()
    for _ in range(msg_cnt):
        msg = await consumer.getone()
        print(f"Message from partition [{msg.partition}]: {msg.value}")
    await consumer.stop()

if len(sys.argv) < 3:
    print("usage: consumer.py <group_id> <wait messages count>")
    sys.exit(1)
group_id = sys.argv[1]
msg_cnt = int(sys.argv[2])

asyncio.run(consume(group_id, msg_cnt))
```

Run example scripts:

- Creating topic `some-topic` with 2 partitions using standard Kafka utility:

```
bin/kafka-topics.sh --create \
  --zookeeper localhost:2181 \
  --replication-factor 1 \
  --partitions 2 \
  --topic some-topic
```

- terminal#1:

```
python3 consumer.py TEST_GROUP 2
```

- terminal#2:

```
python3 consumer.py TEST_GROUP 2
```

- terminal#3:

```
python3 consumer.py OTHER_GROUP 4
```

- terminal#4:

```
python3 producer.py 0 'message #1'
python3 producer.py 0 'message #2'
python3 producer.py 1 'message #3'
python3 producer.py 1 'message #4'
```

Output:

- terminal#1:

```
Message from partition [0]: b'message #1'

Message from partition [0]: b'message #2'
```

- terminal#2:

```
Message from partition [1]: b'message #3'

Message from partition [1]: b'message #4'
```

- terminal#3:

```
Message from partition [1]: b'message #3'

Message from partition [1]: b'message #4'

Message from partition [0]: b'message #1'

Message from partition [0]: b'message #2'
```

4.5.4 Custom partitioner

If you consider using partitions as a logical entity, rather then purely for load-balancing, you may need to have more control over routing messages to partitions. By default hashing algorithms are used.

Producer

```
import asyncio
import random
from aiokafka import AIOKafkaProducer

def my_partitioner(key, all_partitions, available_partitions):
    if key == b'first':
        return all_partitions[0]
    elif key == b'last':
```

(continues on next page)

(continued from previous page)

```
        return all_partitions[-1]
    return random.choice(all_partitions)

async def produce_one(producer, key, value):
    future = await producer.send('foobar', value, key=key)
    resp = await future
    print("'%s' produced in partition: %i"%(value.decode(), resp.partition))

async def produce_task():
    producer = AIOKafkaProducer(
        bootstrap_servers='localhost:9092',
        partitioner=my_partitioner)

    await producer.start()
    await produce_one(producer, b'last', b'1')
    await produce_one(producer, b'some', b'2')
    await produce_one(producer, b'first', b'3')
    await producer.stop()

asyncio.run(produce_task())
```

Output (topic foobar has 10 partitions):

```
>>> python3 producer.py
'1' produced in partition: 9
'2' produced in partition: 6
'3' produced in partition: 0
```

4.5.5 Using SSL with aiokafka

An example of SSL usage with **aiokafka**. Please read *SSL Authentication* for more information.

```
import asyncio
from aiokafka import AIOKafkaProducer, AIOKafkaConsumer
from aiokafka.helpers import create_ssl_context
from kafka.common import TopicPartition

context = create_ssl_context(
    cafile="./ca-cert",  # CA used to sign certificate.
                         # `CARoot` of JKS store container
    certfile="./cert-signed",  # Signed certificate
    keyfile="./cert-key",  # Private Key file of `certfile` certificate
    password="123123"
)

async def produce_and_consume():
    # Produce
    producer = AIOKafkaProducer(
        bootstrap_servers='localhost:9093',
        security_protocol="SSL", ssl_context=context)
```

(continues on next page)

(continued from previous page)

```
    await producer.start()
    try:
        msg = await producer.send_and_wait(
            'my_topic', b"Super Message", partition=0)
    finally:
        await producer.stop()

    consumer = AIOKafkaConsumer(
        "my_topic", bootstrap_servers='localhost:9093',
        security_protocol="SSL", ssl_context=context)
    await consumer.start()
    try:
        consumer.seek(TopicPartition('my_topic', 0), msg.offset)
        fetch_msg = await consumer.getone()
    finally:
        await consumer.stop()

    print("Success", msg, fetch_msg)

if __name__ == "__main__":
    asyncio.run(produce_and_consume())
```

Output:

```
>>> python3 ssl_consume_produce.py
Success RecordMetadata(topic='my_topic', partition=0, topic_
↪partition=TopicPartition(topic='my_topic', partition=0), offset=32)␣
↪ConsumerRecord(topic='my_topic', partition=0, offset=32, timestamp=1479393347381,␣
↪timestamp_type=0, key=None, value=b'Super Message', checksum=469650252, serialized_key_
↪size=-1, serialized_value_size=13)
```

4.5.6 Local state and storing offsets outside of Kafka

While the default for Kafka applications is storing commit points in Kafka's internal storage, you can disable that and use *seek()* to move to stored points. This makes sense if you want to store offsets in the same system as results of computations (filesystem in example below). But that said, you will still probably want to use the coordinated consumer groups feature.

This example shows extensive usage of *ConsumerRebalanceListener* to control what's done before and after rebalance's.

Local State consumer:

```
import asyncio
from aiokafka import AIOKafkaConsumer, ConsumerRebalanceListener
from aiokafka.errors import OffsetOutOfRangeError


import json
import pathlib
from collections import Counter
```

(continues on next page)

(continued from previous page)

```
FILE_NAME_TMPL = "/tmp/my-partition-state-{tp.topic}-{tp.partition}.json"


class RebalanceListener(ConsumerRebalanceListener):

    def __init__(self, consumer, local_state):
        self.consumer = consumer
        self.local_state = local_state

    async def on_partitions_revoked(self, revoked):
        print("Revoked", revoked)
        self.local_state.dump_local_state()

    async def on_partitions_assigned(self, assigned):
        print("Assigned", assigned)
        self.local_state.load_local_state(assigned)
        for tp in assigned:
            last_offset = self.local_state.get_last_offset(tp)
            if last_offset < 0:
                await self.consumer.seek_to_beginning(tp)
            else:
                self.consumer.seek(tp, last_offset + 1)


class LocalState:

    def __init__(self):
        self._counts = {}
        self._offsets = {}

    def dump_local_state(self):
        for tp in self._counts:
            fpath = pathlib.Path(FILE_NAME_TMPL.format(tp=tp))
            with fpath.open("w+") as f:
                json.dump({
                    "last_offset": self._offsets[tp],
                    "counts": dict(self._counts[tp])
                }, f)

    def load_local_state(self, partitions):
        self._counts.clear()
        self._offsets.clear()
        for tp in partitions:
            fpath = pathlib.Path(FILE_NAME_TMPL.format(tp=tp))
            state = {
                "last_offset": -1,  # Non existing, will reset
                "counts": {}
            }
            if fpath.exists():
                with fpath.open("r+") as f:
                    try:
                        state = json.load(f)
```

(continues on next page)

(continued from previous page)

```
                    except json.JSONDecodeError:
                        pass
                self._counts[tp] = Counter(state['counts'])
                self._offsets[tp] = state['last_offset']

    def add_counts(self, tp, counts, last_offset):
        self._counts[tp] += counts
        self._offsets[tp] = last_offset

    def get_last_offset(self, tp):
        return self._offsets[tp]

    def discard_state(self, tps):
        for tp in tps:
            self._offsets[tp] = -1
            self._counts[tp] = Counter()


async def save_state_every_second(local_state):
    while True:
        try:
            await asyncio.sleep(1)
        except asyncio.CancelledError:
            break
        local_state.dump_local_state()


async def consume():
    consumer = AIOKafkaConsumer(
        bootstrap_servers='localhost:9092',
        group_id="my_group",           # Consumer must be in a group to commit
        enable_auto_commit=False,      # Will disable autocommit
        auto_offset_reset="none",
        key_deserializer=lambda key: key.decode("utf-8") if key else "",
    )
    await consumer.start()

    local_state = LocalState()
    listener = RebalanceListener(consumer, local_state)
    consumer.subscribe(topics=["test"], listener=listener)

    save_task = asyncio.create_task(save_state_every_second(local_state))

    try:

        while True:
            try:
                msg_set = await consumer.getmany(timeout_ms=1000)
            except OffsetOutOfRangeError as err:
                # This means that saved file is outdated and should be
                # discarded
                tps = err.args[0].keys()
```

(continues on next page)

(continued from previous page)

```
                local_state.discard_state(tps)
                await consumer.seek_to_beginning(*tps)
                continue

            for tp, msgs in msg_set.items():
                counts = Counter()
                for msg in msgs:
                    print("Process", tp, msg.key)
                    counts[msg.key] += 1
                local_state.add_counts(tp, counts, msg.offset)

    finally:
        await consumer.stop()
        save_task.cancel()
        await save_task


if __name__ == "__main__":
    asyncio.run(consume())
```

There are several points of interest in this example:

- We implement `RebalanceListener` to dump all counts and offsets before rebalances. After rebalances we load them from the same files. It's a kind of cache to avoid re-reading all messages.
- We control offset reset policy manually by setting `auto_offset_reset="none"`. We need it to catch *OffsetOutOfRangeError* so we can clear cache if files were old and such offsets don't exist anymore in Kafka.
- As we count `keys` here, those will always be partitioned to the same partition on produce. We will not have duplicate counts in different files.

Output for 1st consumer:

```
>>> python examples/local_state_consumer.py
Revoked set()
Assigned {TopicPartition(topic='test', partition=0), TopicPartition(topic='test',␣
↪partition=1), TopicPartition(topic='test', partition=2)}
Heartbeat failed for group my_group because it is rebalancing
Revoked {TopicPartition(topic='test', partition=0), TopicPartition(topic='test',␣
↪partition=1), TopicPartition(topic='test', partition=2)}
Assigned {TopicPartition(topic='test', partition=0), TopicPartition(topic='test',␣
↪partition=2)}
Process TopicPartition(topic='test', partition=2) 123
Process TopicPartition(topic='test', partition=2) 9999
Process TopicPartition(topic='test', partition=2) 1111
Process TopicPartition(topic='test', partition=0) 4444
Process TopicPartition(topic='test', partition=0) 123123
Process TopicPartition(topic='test', partition=0) 5555
Process TopicPartition(topic='test', partition=2) 88891823
Process TopicPartition(topic='test', partition=2) 2
```

Output for 2nd consumer:

```
>>> python examples/local_state_consumer.py
Revoked set()
Assigned {TopicPartition(topic='test', partition=1)}
Process TopicPartition(topic='test', partition=1) 321
Process TopicPartition(topic='test', partition=1) 777
```

Those create such files as a result:

```
>>> cat /tmp/my-partition-state-test-0.json && echo
{"last_offset": 4, "counts": {"123123": 1, "4444": 1, "321": 2, "5555": 1}}
```

4.5.7 Batch producer

If your application needs precise control over batch creation and submission and you're willing to forego the niceties of automatic serialization and partition selection, you may use the simple `create_batch()` and `send_batch()` interface.

Producer

```
import asyncio
import random
from aiokafka.producer import AIOKafkaProducer

async def send_many(num):
    topic  = "my_topic"
    producer = AIOKafkaProducer()
    await producer.start()

    batch = producer.create_batch()

    i = 0
    while i < num:
        msg = ("Test message %d" % i).encode("utf-8")
        metadata = batch.append(key=None, value=msg, timestamp=None)
        if metadata is None:
            partitions = await producer.partitions_for(topic)
            partition = random.choice(tuple(partitions))
            await producer.send_batch(batch, topic, partition=partition)
            print("%d messages sent to partition %d"
                  % (batch.record_count(), partition))
            batch = producer.create_batch()
            continue
        i += 1
    partitions = await producer.partitions_for(topic)
    partition = random.choice(tuple(partitions))
    await producer.send_batch(batch, topic, partition=partition)
    print("%d messages sent to partition %d"
          % (batch.record_count(), partition))
    await producer.stop()

asyncio.run(send_many(1000))
```

Output (topic `my_topic` has 3 partitions):

```
>>> python3 batch_produce.py
329 messages sent to partition 2
327 messages sent to partition 0
327 messages sent to partition 0
17 messages sent to partition 1
```

4.5.8 Transactional Consume-Process-Produce

If you have a pattern where you want to consume from one topic, process data and produce to a different one, you would really like to do it with using Transactional Producer. In the example below we read from `IN_TOPIC`, process data and produce the resut to `OUT_TOPIC` in a transactional manner.

```
import asyncio
from collections import defaultdict, Counter

from aiokafka import TopicPartition, AIOKafkaConsumer, AIOKafkaProducer


IN_TOPIC = "in_topic"
GROUP_ID = "processing-group"
OUT_TOPIC = "out_topic"
TRANSACTIONAL_ID = "my-txn-id"
BOOTSTRAP_SERVERS = "localhost:9092"

POLL_TIMEOUT = 60_000


def process_batch(msgs):
    # Group by key do simple count sampling by a minute window
    buckets_by_key = defaultdict(Counter)
    for msg in msgs:
        timestamp = (msg.timestamp // 60_000) * 60
        buckets_by_key[msg.key][timestamp] += 1

    res = []
    for key, counts in buckets_by_key.items():
        for timestamp, count in counts.items():
            value = str(count).encode()
            res.append((key, value, timestamp))

    return res


async def transactional_process():
    consumer = AIOKafkaConsumer(
        IN_TOPIC,
        bootstrap_servers=BOOTSTRAP_SERVERS,
        enable_auto_commit=False,
        group_id=GROUP_ID,
        isolation_level="read_committed"  # <-- This will filter aborted txn's
    )
```

(continues on next page)

(continued from previous page)

```
    await consumer.start()

    producer = AIOKafkaProducer(
        bootstrap_servers=BOOTSTRAP_SERVERS,
        transactional_id=TRANSACTIONAL_ID
    )
    await producer.start()

    try:
        while True:
            msg_batch = await consumer.getmany(timeout_ms=POLL_TIMEOUT)

            async with producer.transaction():
                commit_offsets = {}
                in_msgs = []
                for tp, msgs in msg_batch.items():
                    in_msgs.extend(msgs)
                    commit_offsets[tp] = msgs[-1].offset + 1

                out_msgs = process_batch(in_msgs)
                for key, value, timestamp in out_msgs:
                    await producer.send(
                        OUT_TOPIC, value=value, key=key,
                        timestamp_ms=int(timestamp * 1000)
                    )
                # We commit through the producer because we want the commit
                # to only succeed if the whole transaction is done
                # successfully.
                await producer.send_offsets_to_transaction(
                    commit_offsets, GROUP_ID)
    finally:
        await consumer.stop()
        await producer.stop()


if __name__ == "__main__":
    asyncio.run(transactional_process())
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

k

INDEX

T

U

V