# CONTENTS

1 Getting started
   1.1 AIOKafkaConsumer ............................................. 3
   1.2 AIOKafkaProducer ............................................. 3

2 Installation
   2.1 Optional LZ4 install ............................................ 5
   2.2 Optional Snappy install ....................................... 5
   2.3 Optional GSSAPI install ....................................... 6

3 Source code

4 Authors and License
   4.1 Producer client .............................................. 9
   4.1.1 Message buffering ......................................... 9
   4.1.2 Retries and Message acknowledgement ..................... 10
   4.1.3 Idempotent produce ....................................... 10
   4.1.4 Transactional producer .................................... 11
   4.1.5 Returned RecordMetadata object .......................... 12
   4.1.6 Direct batch control ..................................... 12
   4.2 Consumer client ............................................. 13
   4.2.1 Offsets and Consumer Position ............................ 13
   4.2.2 Consumer Groups and Topic Subscriptions ............... 17
   4.2.3 Detecting Consumer Failures ............................... 21
   4.3 Difference between aiokafka and kafka-python ............ 21
   4.3.1 Why do we need another library? .......................... 21
   4.3.2 API differences and rationale ............................. 21
   4.4 API Documentation .......................................... 23
   4.4.1 AIOKafkaProducer class ................................... 23
   4.4.2 AIOKafkaConsumer class ................................... 26
   4.4.3 Helpers ................................................ 32
   4.4.4 Abstracts ................................................ 32
   4.4.5 SSL Authentication ....................................... 33
   4.4.6 SASL Authentication ....................................... 33
   4.4.7 Error handling ............................................ 34
   4.5 Examples .................................................. 34
   4.5.1 Serialization and compression .............................. 34
   4.5.2 Manual commit ............................................ 36
   4.5.3 Group consumer ........................................... 36
   4.5.4 Custom partitioner ........................................ 38
   4.5.5 Using SSL with aiokafka ................................... 39
| 4.5.6 | Local state and storing offsets outside of Kafka | 40 |
| 4.5.7 | Batch producer | 43 |
| 4.5.8 | Transactional Consume-Process- Produce | 44 |

5 Indices and tables 47

Python Module Index 49

Index 51
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.
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:

```python
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:

```python
from aiokafka import AIOKafkaProducer
import asyncio

async def send_one():
    producer = AIOKafkaProducer(
```

(continues on next page)
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.
2.1 Optional LZ4 install

To enable LZ4 compression/decompression, install lz4tools and xxhash:

```
>>> pip3 install lz4tools
>>> pip3 install xxhash
```

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-XXXXXXX.tar.gz
cd google-snappy-X.X.X-X-XXXXXXX
./configure
make
sudo make install
```

2. Install the python-snappy module
pip3 install python-snappy

For Windows the easiest way is to fetch a precompiled wheel from http://www.lfd.uci.edu/~gohlke/pythonlibs/#python-snappy

### 2.3 Optional GSSAPI install

To enable SASL authentication with GSSAPI you need to install `gssapi`:

```bash
>>> pip3 install gssapi
```
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 Continuous Integration.
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:

```python
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, Producer 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 send_and_wait shortcut:

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

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:

```python
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`:

```python
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.
- In 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:

```python
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 retrievable 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 *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:

```python
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 containing fields:

- **offset** - unique offset of the message in this partition. See `Offsets and Consumer Position` for more details on offsets.
- **topic** - `string` topic name
- **partition** - `int` partition number
- **timestamp** - `int` timestamp in epoch milliseconds (from Jan 1 1970 UTC)
- **timestamp_type** - `int` if broker respected the timestamp passed to `send()` 0 will be returned (CreateTime). If Broker set it’s own timestamp 1 will be returned (LogAppendTime).

**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:

```python
# 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, `record.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:

```python
consumer = aiokafka.AIOKafkaConsumer(
    "my_topic",
    bootstrap_servers='localhost:9092'
)
await consumer.start()
try:
    async for msg in consumer:
        print("{}: {}:{}: 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 `key_deserializer` and `value_deserializer` configuration if you need to decode them.

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

Consumer 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:

```python
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 `consumer.commit()` and `consumer.committed()` API you need to set `group_id` to something other than `None`. See Consumer Groups and Topic Subscriptions 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:

```python
c 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:

```python
c 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
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:

```python
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 `commit(offsets)` 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:

```python
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 `consumer.seek()`. For example, you can re-consume records:

```python
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 `consumer.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:

```python
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)

```python
async for msg in consumer:
    key = msg.key.decode("utf-8")
    counts[key] += 1
    value = json.dumps(
        
        "count": counts[key],
        "offset": msg.offset
    )
    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 consumer.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 **Consumer** instances sharing the same **group_id** will be part of the same **Consumer Group**:

```python
# 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 consumer.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 aiokafka.AIOKafkaConsumer.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:

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

class MyRebalancer(aiokafka.ConsumerRebalanceListener):
    async def on_partitions_revoked(self, revoked):
        async with self.lock:
            pass
    async def on_partitions_assigned(self, assigned):
        pass

async def main():
    consumer.subscribe("topic", listener=MyRebalancer())
    while True:
        async with self.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**

Consumer 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:

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

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

Here **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:

```python
counter = 0
for i in range(10):
    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)
```

**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:

```python
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 consumer.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:

```python
consumer = aiokafka.AIOKafkaConsumer(
    "my_topic",
    bootstrap_servers='localhost:9092',
    isolation_level="read_committed"
)
await consumer.start()
async for msg in consumer:  # Only read committed transactions
    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(*partitions) and end_offsets(partitions) 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:

```python
async for msg in consumer:  # Only read committed transactions
    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. These 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 Group. 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 consumer.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 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 \textit{poll()} method

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

\begin{itemize}
  \item Socket polling using \texttt{epoll}, \texttt{kqueue} or other available API of your OS.
  \item Ensures liveliness of a Consumer Group
  \item Does autocommit
\end{itemize}

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 \texttt{AIOKafkaConsumer} class exposes this interface under the name \texttt{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 \texttt{poll()} was called. This lead to a lot of issues (reasons for KIP-41 and KIP-62 proposals) and workarounds using \texttt{pause()} and \texttt{poll(0)} for heartbeats. After Java client and kafka-python also changed the behaviour to a background Thread sending, that mitigated most issues. \texttt{aiokafka} delegates heartbeating to a background \texttt{Task} and will send heartbeats to Coordinator as long as the \textit{event loop} is running. This behaviour is very similar to Java client, with the exception of no heartbeats on long CPU bound methods.

But \texttt{aiokafka} also performs group rebalancing in the same background Task. This means, that the processing time between \texttt{getmany} calls actually does not affect rebalancing. KIP-62 proposed to provide \texttt{max.poll.interval.ms} as the configuration for both \textit{rebalance timeout} and \textit{consumer processing timeout}. In \texttt{aiokafka} it does not make much sense, as those 2 are not related, so we added both configurations (\texttt{rebalance_timeout_ms} and \texttt{max_poll_interval_ms}). It is quite critical to provide \texttt{ConsumerRebalanceListener} if you need to control rebalance start and end moments. In that case set the \texttt{rebalance_timeout_ms} to the maximum time your application can spend waiting in the callback. If your callback waits for the last \texttt{getmany} result to be processed, it is safe to set this value to \texttt{max_poll_interval_ms}, same as in Java client.

Prefetching is more sophisticated

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

\begin{itemize}
  \item in \texttt{poll()} call if we don’t have enough data stored to satisfy another \texttt{poll()}
  \item in the \texttt{iterator} interface if we have processed \textit{nearly} all data.
\end{itemize}

A very simplified version would be:

\begin{verbatim}
def poll()
    max_records = self.config['max_poll_records']
    records = consumer.fetched_records(max_records)
    if not consumer.has_enough_records(max_records)
        consumer.send_fetches()  # prefetch another batch
    return records
\end{verbatim}

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 \textbf{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 \texttt{aiokafka} tries to do prefetches \textbf{per partition}. For example, if we processed all data pending for a partition in \texttt{iterator} interface, \texttt{aiokafka} will \texttt{try} to prefetch new data right away. The same interface could be built on top of \texttt{kafka-python's pause API}, but would require \textbf{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 AIOKafkaProducer class

```python
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** – ‘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** *(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’, 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**. This setting defaults to 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, SASL_PLAINTEXT, SASL_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 `Abstracts`. 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** *(kafka.oauth.abstract.AbstractTokenProvider)* – OAuthBearer token provider instance. (See `kafka.oauth.abstract`). Default: None

---

**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
4.4.2 AIOKafkaConsumer class

class aiokafka.AIOKafkaConsumer(*topics, loop=None, bootstrap_servers=localhost,
client_id='aiokafka-0.7.2', 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** *(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** – ‘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 OffsetOutOfRange 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., `consumer.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 rebalancing. 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, SASL_PLAINTEXT, SASL_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 Abstracts. 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 `readcommitted`, `consumer.getmany()` will only return transactional messages which have been committed. If set to `read_uncommitted` (the default), `consumer.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, `consumer.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 (kafka.oauth.abstract.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 TopicPartitions to this consumer.

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

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

**Raises** `IllegalStateException` – 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** `{TopicPartition, …}`

**Return type** `set`

### 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`
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_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

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

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
• IllegalState Error – partition is not currently assigned
Changed in version 0.4.0: Changed `AssertionError` to `IllegalStateError` and `ValueError` in respective cases.

**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 topic subscription.

**Returns** `{topic, ...}`

**Return type** `frozenset`

**unsubscribe()**

Unsubscribe from all topics and clear all assigned partitions.

**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** *(TopicPartition)* – Partitions to pause.
paused()
    Get the partitions that were previously paused using pause().

    Returns {partition (TopicPartition), ...}

    Return type set

resume (*partitions)
    Resume fetching from the specified (paused) partitions.

    Parameters *partitions (TopicPartition) – Partitions to resume.

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.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.

### 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:

```python
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 consumption 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`.

4.5 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 `msg.value` and `msg.key` attributes of returned `msg` instances are `bytes`. You can use custom serializer/deserializer hooks to operate on objects instead of bytes in those attributes.

Producer

```python
import json
import asyncio
from aiokafka import AIOKafkaProducer

def serializer(value):
    return json.dumps(value).encode()

async def produce():
    producer = AIOKafkaProducer()
    ```
```python
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**

```python
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:

```python
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()

    for i 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 (though 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:

```python
import sys
import asyncio
from aiokafka import AIOKafkaProducer

async def produce(value, partition):
    producer = AIOKafkaProducer(bootstrap_servers='localhost:9092')
    ```

(continues on next page)
async def producer():
    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:

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

terminal#1:

```bash
>>> python3 consumer.py TEST_GROUP 2
```

terminal#2:

```bash
>>> python3 consumer.py TEST_GROUP 2
```

terminal#3:

4.5. Examples

37
>>> 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

```python
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':
        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("'{}' produced in partition: {}\r\n")
    (value.decode(), resp.partition)

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

(continues on next page)```
```
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 Abstracts for more information.

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

certfile='./cert-signed',
keyfile='./cert-key',
password='123123'

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

    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", ssi_context=context)
    await consumer.start()
    try:
        consumer.seek(TopicPartition('my_topic', 0), msg.offset)
        fetch_msg = await consumer.getone()
    finally:
        await consumer.stop()
```

(continues on next page)
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:

```python
import asyncio
from aiokafka import AIOKafkaConsumer, ConsumerRebalanceListener
from aiokafka.errors import OffsetOutOfRangeError
import json
import pathlib
from collections import Counter

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:
```
```python
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)
                    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
```

4.5. Examples
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()
                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
>>> python examples/local_state_consumer.py
Revoke 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
Revoke {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
>>> python examples/local_state_consumer.py
Revoke 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:

```python
>>> 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**

```python
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)
```

(continues on next page)
if metadata is None:
    partitions = await producer.partitions_for(topic)
    partition = random.choice(tuple(partitions))
    await producer.send_batch(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, 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):

```bash
>>> 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 result to **OUT_TOPIC** in a transactional manner.

```python
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 = []
```

(continues on next page)
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
    )
    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

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>aiokafka.helpers, 32</td>
</tr>
</tbody>
</table>
INDEX

A
aiokafka.helpers
module, 32
AIOKafkaConsumer (class in aiokafka), 26
AIOKafkaProducer (class in aiokafka), 23
assign() (aiokafka.AIOKafkaConsumer method), 29
assignment() (aiokafka.AIOKafkaConsumer method), 29

C
ConsumerRebalanceListener (class in aiokafka),
32
create_batch() (aiokafka.AIOKafkaProducer
method), 25
create_ssl_context() (in module
aiokafka.helpers), 32

H
highwater() (aiokafka.AIOKafkaConsumer method),
29

L
last_poll_timestamp()
(aiokafka.AIOKafkaConsumer method),
30
last_stable_offset()
(aiokafka.AIOKafkaConsumer method),
30

M
module
aiokafka.helpers, 32

P
partitions_for_topic()
(aiokafka.AIOKafkaConsumer method),
29
pause() (aiokafka.AIOKafkaConsumer method), 31
paused() (aiokafka.AIOKafkaConsumer method), 31

R
resume() (aiokafka.AIOKafkaConsumer method), 32

S
seek() (aiokafka.AIOKafkaConsumer method), 30
subscribe() (aiokafka.AIOKafkaConsumer method),
31
subscription() (aiokafka.AIOKafkaConsumer
method), 31

U
unsubscribe() (aiokafka.AIOKafkaConsumer
method), 31