kafka-python Documentation

Release 1.0.0

Dana Powers

February 16, 2016

Contents

Python client for the Apache Kafka distributed stream processing system. kafka-python is designed to function much like the official java client, with a sprinkling of pythonic interfaces (e.g., consumer iterators).

kafka-python is best used with 0.9 brokers, but is backwards-compatible with older versions (to 0.8.0). Some features will only be enabled on newer brokers, however; for example, fully coordinated consumer groups – i.e., dynamic partition assignment to multiple consumers in the same group – requires use of 0.9 kafka brokers. Supporting this feature for earlier broker releases would require writing and maintaining custom leadership election and membership / health check code (perhaps using zookeeper or consul). For older brokers, you can achieve something similar by manually assigning different partitions to each consumer instance with config management tools like chef, ansible, etc. This approach will work fine, though it does not support rebalancing on failures. See Compatibility for more details.

Please note that the master branch may contain unreleased features. For release documentation, please see readthedocs and/or python's inline help.

```
>>> pip install kafka-python
```

CHAPTER 1

KafkaConsumer

`KafkaConsumer` is a high-level message consumer, intended to operate as similarly as possible to the official 0.9 java client. Full support for coordinated consumer groups requires use of kafka brokers that support the 0.9 Group APIs.

See KafkaConsumer for API and configuration details.

The consumer iterator returns ConsumerRecords, which are simple namedtuples that expose basic message attributes: topic, partition, offset, key, and value:

```
>>> from kafka import KafkaConsumer
>>> consumer = KafkaConsumer('my_favorite_topic')
>>> for msg in consumer:
...     print (msg)
```

```
>>> # manually assign the partition list for the consumer
>>> from kafka import TopicPartition
>>> consumer = KafkaConsumer(bootstrap_servers='localhost:1234')
>>> consumer.assign([TopicPartition('foobar', 2)])
>>> msg = next(consumer)
```

```
>>> # Deserialize msgpack-encoded values
>>> consumer = KafkaConsumer(value_deserializer=msgpack.dumps)
>>> consumer.subscribe(['msgpackfoo'])
>>> for msg in consumer:
...     msg = next(consumer)
...     assert isinstance(msg.value, dict)
```

CHAPTER 2

KafkaProducer

KafkaProducer is a high-level, asynchronous message producer. The class is intended to operate as similarly as possible to the official java client. See KafkaProducer for more details.

```
>>> from kafka import KafkaProducer
>>> producer = KafkaProducer(bootstrap_servers='localhost:1234')
>>> producer.send('foobar', b'some_message_bytes')
```

```
>>> # Blocking send
>>> producer.send('foobar', b'another_message').get(timeout=60)
```

```
>>> # Use a key for hashed-partitioning
>>> producer.send('foobar', key=b'foo', value=b'bar')
```

```
>>> # Serialize json messages
>>> import json
>>> producer = KafkaProducer(value_serializer=json.loads)
>>> producer.send('fizzbuzz', {'foo': 'bar'})
```

```
>>> # Serialize string keys
>>> producer = KafkaProducer(key_serializer=str.encode)
>>> producer.send('flipflap', key='ping', value=b'1234')
```

```
>>> # Compress messages
>>> producer = KafkaProducer(compression_type='gzip')
>>> for i in range(1000):
...     producer.send('foobar', b'msg %d' % i)
```

CHAPTER 3

Compression

kafka-python supports gzip compression/decompression natively. To produce or consume lz4 compressed messages, you must install lz4tools and xxhash (modules may not work on python2.6). To enable snappy, install python-snappy (also requires snappy library). See Installation for more information.

CHAPTER 4

Protocol

A secondary goal of kafka-python is to provide an easy-to-use protocol layer for interacting with kafka brokers via the python repl. This is useful for testing, probing, and general experimentation. The protocol support is leveraged to enable a `check_version()` method that probes a kafka broker and attempts to identify which version it is running (0.8.0 to 0.9).

CHAPTER 5

Low-level

Legacy support is maintained for low-level consumer and producer classes, SimpleConsumer and SimpleProducer.

5.1 Usage

5.1.1 KafkaConsumer

```
from kafka import KafkaConsumer

# To consume latest messages and auto-commit offsets
consumer = KafkaConsumer('my-topic',
                         group_id='my-group',
                         bootstrap_servers=['localhost:9092'])
for message in consumer:
    # message value and key are raw bytes -- decode if necessary!
    # e.g., for unicode: `message.value.decode('utf-8')`
    print ("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,
                                          message.offset, message.key,
                                          message.value))

# consume earliest available messages, dont commit offsets
KafkaConsumer(auto_offset_reset='earliest', enable_auto_commit=False)

# consume json messages
KafkaConsumer(value_deserializer=lambda m: json.loads(m.decode('ascii')))

# consume msgpack
KafkaConsumer(value_deserializer=msgpack.unpackb)

# StopIteration if no message after 1sec
KafkaConsumer(consumer_timeout_ms=1000)

# Subscribe to a regex topic pattern
consumer = KafkaConsumer()
consumer.subscribe(pattern='^awesome.*')

# Use multiple consumers in parallel w/ 0.9 kafka brokers
# typically you would run each on a different server / process / CPU
consumer1 = KafkaConsumer('my-topic',
                          group_id='my-group',
                          bootstrap_servers='my.server.com')
```

```
consumer2 = KafkaConsumer('my-topic',
                          group_id='my-group',
                          bootstrap_servers='my.server.com')
```

There are many configuration options for the consumer class. See `KafkaConsumer` API documentation for more details.

5.1.2 KafkaProducer

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers=['broker1:1234'])

# Asynchronous by default
future = producer.send('my-topic', b'raw_bytes')

# Block for 'synchronous' sends
try:
    record_metadata = future.get(timeout=10)
except KafkaError:
    # Decide what to do if produce request failed...
    log.exception()
    pass

# Successful result returns assigned partition and offset
print (record_metadata.topic)
print (record_metadata.partition)
print (record_metadata.offset)

# produce keyed messages to enable hashed partitioning
producer.send('my-topic', key=b'foo', value=b'bar')

# encode objects via msgpack
producer = KafkaProducer(value_serializer=msgpack.dumps)
producer.send('msgpack-topic', {'key': 'value'})

# produce json messages
producer = KafkaProducer(value_serializer=lambda m: json.dumps(m).encode('ascii'))
producer.send('json-topic', {'key': 'value'})

# configure multiple retries
producer = KafkaProducer(retries=5)
```

5.2 kafka-python API

5.2.1 KafkaConsumer

class kafka.**KafkaConsumer**(**topics*, ***configs*)

Consume 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 (requires 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.

Keyword Arguments

- **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: 'kafka-python-{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: 'kafka-python-default-group'
- **key_deserializer** (*callable*) – Any callable that takes a raw message key and returns a deserialized key.
- **value_deserializer** (*callable*) – 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_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.
- **request_timeout_ms** (*int*) – Client request timeout in milliseconds. Default: 40000.
- **retry_backoff_ms** (*int*) – Milliseconds to backoff when retrying on errors. Default: 100.
- **reconnect_backoff_ms** (*int*) – The amount of time in milliseconds to wait before attempting to reconnect to a given host. Default: 50.
- **max_in_flight_requests_per_connection** (*int*) – Requests are pipelined to kafka brokers up to this number of maximum requests per broker connection. Default: 5.
- **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. Any ofther value will raise the exception. 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.

- **default_offset_commit_callback** (*callable*) – called as callback(offsets, response) response will be either an Exception or a OffsetCommitResponse struct. This callback can be used to trigger custom actions when a commit request completes.
- **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. Default: [RoundRobinPartitionAssignor]
- **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
- **session_timeout_ms** (*int*) – The timeout used to detect failures when using Kafka's group managementment facilities. Default: 30000
- **send_buffer_bytes** (*int*) – The size of the TCP send buffer (SO_SNDBUF) to use when sending data. Default: 131072
- **receive_buffer_bytes** (*int*) – The size of the TCP receive buffer (SO_RCVBUF) to use when reading data. Default: 32768
- **consumer_timeout_ms** (*int*) – number of millisecond to throw a timeout exception to the consumer if no message is available for consumption. Default: -1 (dont throw exception)
- **api_version** (*str*) – specify which kafka API version to use. 0.9 enables full group coordination features; 0.8.2 enables kafka-storage offset commits; 0.8.1 enables zookeeper-storage offset commits; 0.8.0 is what is left. If set to 'auto', will attempt to infer the broker version by probing various APIs. Default: auto

---

Note: Configuration parameters are described in more detail at https://kafka.apache.org/090/configuration.html#newconsumerconfigs

---

assign(*partitions*)

Manually assign a list of TopicPartitions to this consumer.

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

Raises IllegalStateError – if consumer has already called subscribe()

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

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

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 TopicPartitions 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 none if the assignment hasn't happened yet, or if the partitions are in the process of being reassigned).

Returns {TopicPartition, ...}

Return type set

close()
Close the consumer, waiting indefinitely for any needed cleanup.

commit(*offsets=None*)
Commit offsets to kafka, blocking until success or error

This commits offsets only to Kafka. The offsets committed using this API will be used on the first fetch after every rebalance and also on startup. As such, if you need to store offsets in anything other than Kafka, this API should not be used. To avoid re-processing the last message read if a consumer is restarted, the committed offset should be the next message your application should consume, i.e.: last_offset + 1.

Blocks until either the commit succeeds or an unrecoverable error is encountered (in which case it is thrown to the caller).

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

Parameters **offsets** (*dict, optional*) – {TopicPartition: OffsetAndMetadata} dict to commit with the configured group_id. Defaults to current consumed offsets for all subscribed partitions.

commit_async(*offsets=None*, *callback=None*)
Commit offsets to kafka asynchronously, optionally firing callback

This commits offsets only to Kafka. The offsets committed using this API will be used on the first fetch after every rebalance and also on startup. As such, if you need to store offsets in anything other than Kafka, this API should not be used. To avoid re-processing the last message read if a consumer is restarted, the committed offset should be the next message your application should consume, i.e.: last_offset + 1.

This is an asynchronous call and will not block. Any errors encountered are either passed to the callback (if provided) or discarded.

Parameters

- **offsets** (*dict, optional*) – {TopicPartition: OffsetAndMetadata} dict to commit with the configured group_id. Defaults to current consumed offsets for all subscribed partitions.
- **callback** (*callable, optional*) – called as callback(offsets, response) with response as either an Exception or a OffsetCommitResponse struct. This callback can be used to trigger custom actions when a commit request completes.

Returns kafka.future.Future

committed(*partition*)
Get the last committed offset for the given partition

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

This call may block to do a remote call if the partition in question isn't assigned to this consumer or if the consumer hasn't yet initialized its cache of committed offsets.

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

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

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 in FetchResponse messages, so will not be available if not FetchRequests have been sent for this partition yet.

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

Returns offset if available

Return type int or None

partitions_for_topic(*topic*)
Get metadata about the partitions for a given topic.

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

Returns partition ids

Return type set

pause(**partitions*)
Suspend fetching from the requested partitions.

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

poll(*timeout_ms=0*)
Fetch data from assigned topics / partitions.

Records are fetched and returned in batches by topic-partition. On each poll, consumer will try to use the last consumed offset as the starting offset and fetch sequentially. The last consumed offset can be manually set through seek(partition, offset) or automatically set as the last committed offset for the subscribed list of partitions.

Incompatible with iterator interface – use one or the other, not both.

Parameters **timeout_ms** (*int, optional*) – milliseconds spent waiting in poll if data is not available in the buffer. If 0, returns immediately with any records that are available currently in the buffer, else returns empty. Must not be negative. Default: 0

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

Return type dict

position(*partition*)
Get the offset of the next record that will be fetched

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

Returns offset

Return type int

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

Parameters ***partitions** (*TopicPartition*) – partitions to resume

seek(*partition*, *offset*)
Manually specify the fetch offset for a TopicPartition.

Overrides the fetch offsets that the consumer will use on the next poll(). If this API is invoked for the same partition more than once, the latest offset will be used on the next poll(). Note that you may lose data if this API is arbitrarily used in the middle of consumption, to reset the fetch offsets.

Parameters

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

Raises AssertionError – if offset is not an int >= 0; or if partition is not currently assigned.

seek_to_beginning(**partitions*)
Seek to the oldest available offset for partitions.

Parameters ***partitions** – optionally provide specific TopicPartitions, otherwise default to all assigned partitions

Raises AssertionError – if any partition is not currently assigned, or if no partitions are assigned

seek_to_end(**partitions*)
Seek to the most recent available offset for partitions.

Parameters ***partitions** – optionally provide specific TopicPartitions, otherwise default to all assigned partitions

Raises AssertionError – if any partition is not currently assigned, or if no partitions are assigned

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()
- `AssertionError` – if neither topics or pattern is provided
- `TypeError` – if listener is not a ConsumerRebalanceListener

subscription()
Get the current topic subscription.

Returns {topic, ...}

Return type set

topics()
Get all topics the user is authorized to view.

Returns topics

Return type set

unsubscribe()
Unsubscribe from all topics and clear all assigned partitions.

5.2.2 KafkaProducer

class kafka.**KafkaProducer**(***configs*)
A Kafka client that publishes records to the Kafka cluster.

The producer is thread safe and sharing a single producer instance across threads will generally be faster than having multiple instances.

The producer consists of a pool of buffer space that holds records that haven't yet been transmitted to the server as well as a background I/O thread that is responsible for turning these records into requests and transmitting them to the cluster.

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 blocking on the full commit of the record, the slowest but most durable setting.

If the request fails, the producer can automatically retry, unless 'retries' is configured to 0. Enabling retries also opens up the possibility of duplicates (see the documentation on message delivery semantics for details: http://kafka.apache.org/documentation.html#semantics).

The producer maintains buffers of unsent records for each partition. These buffers are of a size specified by the 'batch_size' config. Making this larger can result in more batching, but requires more memory (since we will generally have one of these buffers for each active partition).

By default a buffer is available to send immediately even if there is additional unused space in the buffer. However if you want to reduce the number of requests you can set 'linger_ms' to something greater than 0. This will instruct the producer to wait up to that number of milliseconds before sending a request in hope that more records will arrive to fill up the same batch. This is analogous to Nagle's algorithm in TCP. Note that records that arrive close together in time will generally batch together even with linger_ms=0 so under heavy load batching will occur regardless of the linger configuration; however setting this to something larger than 0 can lead to fewer, more efficient requests when not under maximal load at the cost of a small amount of latency.

The buffer_memory controls the total amount of memory available to the producer for buffering. If records are sent faster than they can be transmitted to the server then this buffer space will be exhausted. When the buffer space is exhausted additional send calls will block.

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

Keyword Arguments

- **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: 'kafka-python-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. 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: Wait for leader to write the record to its local log only. Broker 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: Wait for the full set of in-sync replicas to write 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.
- **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.
- **retries** (*int*) – Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries will potentially change

the ordering of records because if two records are sent to a single partition, and the first fails and is retried but the second succeeds, then the second record may appear first. Default: 0.

- **batch_size** (*int*) – Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent. A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). 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, rather than immediately sending out a record the producer will wait for up to the given delay to allow other records to be sent so that the sends can be batched together. This can be thought of as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on the delay for batching: once we get batch_size worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up. This setting defaults to 0 (i.e. no delay). Setting linger_ms=5 would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absense of load. Default: 0.
- **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).
- **buffer_memory** (*int*) – The total bytes of memory the producer should use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the server the producer will block up to max_block_ms, raising an exception on timeout. In the current implementation, this setting is an approximation. Default: 33554432 (32MB)
- **max_block_ms** (*int*) – Number of milliseconds to block during send() when attempting to allocate additional memory before raising an exception. Default: 60000.
- **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
- **retry_backoff_ms** (*int*) – Milliseconds to backoff when retrying on errors. Default: 100.
- **request_timeout_ms** (*int*) – Client request timeout in milliseconds. Default: 30000.
- **receive_buffer_bytes** (*int*) – The size of the TCP receive buffer (SO_RCVBUF) to use when reading data. Default: 32768
- **send_buffer_bytes** (*int*) – The size of the TCP send buffer (SO_SNDBUF) to use when sending data. Default: 131072
- **reconnect_backoff_ms** (*int*) – The amount of time in milliseconds to wait before attempting to reconnect to a given host. Default: 50.

- **max_in_flight_requests_per_connection** (*int*) – Requests are pipelined to kafka brokers up to this number of maximum requests per broker connection. Default: 5.
- **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

---

Note: Configuration parameters are described in more detail at https://kafka.apache.org/090/configuration.html#producerconfigs

---

close(*timeout=None*)
Close this producer.

flush()
Invoking this method makes all buffered records immediately available to send (even if linger_ms is greater than 0) and blocks on the completion of the requests associated with these records. The post-condition of flush() is that any previously sent record will have completed (e.g. Future.is_done() == True). A request is considered completed when either it is successfully acknowledged according to the 'acks' configuration for the producer, or it results in an error.

Other threads can continue sending messages while one thread is blocked waiting for a flush call to complete; however, no guarantee is made about the completion of messages sent after the flush call begins.

partitions_for(*topic*)
Returns set of all known partitions for the topic.

send(*topic*, *value=None*, *key=None*, *partition=None*)
Publish a message to a topic.

Parameters

- **topic** (*str*) – topic where the message will be published
- **value** (*optional*) – message value. Must be type bytes, or be serializable to bytes via configured value_serializer. If value is None, key is required and message acts as a 'delete'. See kafka compaction documentation for more details: http://kafka.apache.org/documentation.html#compaction (compaction requires kafka >= 0.8.1)
- **partition** (*int, optional*) – optionally specify a partition. If not set, the partition will be selected using the configured 'partitioner'.
- **key** (*optional*) – a key to associate with the message. Can be used to determine which partition to send the message to. If partition is None (and producer's partitioner config is left as default), then messages with the same key will be delivered to the same partition (but if key is None, partition is chosen randomly). Must be type bytes, or be serializable to bytes via configured key_serializer.

Returns resolves to RecordMetadata

Return type FutureRecordMetadata

Raises `KafkaTimeoutError` – if unable to fetch topic metadata, or unable to obtain memory buffer prior to configured max_block_ms

5.2.3 KafkaClient

class `kafka.client.`**KafkaClient**(***configs*)
A network client for asynchronous request/response network i/o. This is an internal class used to implement the user-facing producer and consumer clients.

This class is not thread-safe!

add_topic(*topic*)
Add a topic to the list of topics tracked via metadata.

Parameters **topic** (*str*) – topic to track

Returns resolves after metadata request/response

Return type Future

check_version(*node_id=None*, *timeout=2*, *strict=False*)
Attempt to guess the broker version

close(*node_id=None*)
Closes the connection to a particular node (if there is one).

Parameters **node_id** (*int*) – the id of the node to close

connection_delay(*node_id*)
Returns the number of milliseconds to wait, based on the connection state, before attempting to send data. When disconnected, this respects the reconnect backoff time. When connecting, returns 0 to allow non-blocking connect to finish. When connected, returns a very large number to handle slow/stalled connections.

Parameters **node_id** (*int*) – The id of the node to check

Returns The number of milliseconds to wait.

Return type int

in_flight_request_count(*node_id=None*)
Get the number of in-flight requests for a node or all nodes.

Parameters **node_id** (*int, optional*) – a specific node to check. If unspecified, return the total for all nodes

Returns pending in-flight requests for the node, or all nodes if None

Return type int

is_disconnected(*node_id*)
Check whether the node connection has been disconnected failed.

A disconnected node has either been closed or has failed. Connection failures are usually transient and can be resumed in the next ready() call, but there are cases where transient failures need to be caught and re-acted upon.

Parameters **node_id** (*int*) – the id of the node to check

Returns True iff the node exists and is disconnected

Return type bool

is_ready(*node_id*)
Check whether a node is ready to send more requests.

In addition to connection-level checks, this method also is used to block additional requests from being sent during a metadata refresh.

Parameters **node_id** (*int*) – id of the node to check

Returns True if the node is ready and metadata is not refreshing

Return type bool

least_loaded_node()
Choose the node with fewest outstanding requests, with fallbacks.

This method will prefer a node with an existing connection, but will potentially choose a node for which we don't yet have a connection if all existing connections are in use. This method will never choose a node that was disconnected within the reconnect backoff period. If all else fails, the method will attempt to bootstrap again using the bootstrap_servers list.

Returns node_id or None if no suitable node was found

poll(*timeout_ms=None*, *future=None*, *sleep=False*)
Try to read and write to sockets.

This method will also attempt to complete node connections, refresh stale metadata, and run previously-scheduled tasks.

Parameters

- **timeout_ms** (*int, optional*) – maximum amount of time to wait (in ms) for at least one response. Must be non-negative. The actual timeout will be the minimum of timeout, request timeout and metadata timeout. Default: request_timeout_ms
- **future** (*Future, optional*) – if provided, blocks until future.is_done
- **sleep** (*bool*) – if True and there is nothing to do (no connections or requests in flight), will sleep for duration timeout before returning empty results. Default: False.

Returns responses received (can be empty)

Return type list

ready(*node_id*)
Check whether a node is connected and ok to send more requests.

Parameters **node_id** (*int*) – the id of the node to check

Returns True if we are ready to send to the given node

Return type bool

schedule(*task*, *at*)
Schedule a new task to be executed at the given time.

This is "best-effort" scheduling and should only be used for coarse synchronization. A task cannot be scheduled for multiple times simultaneously; any previously scheduled instance of the same task will be cancelled.

Parameters

- **task** (*callable*) – task to be scheduled
- **at** (*float or int*) – epoch seconds when task should run

Returns resolves to result of task call, or exception if raised

Return type Future

send(*node_id*, *request*)
Send a request to a specific node.

Parameters

- **node_id** (*int*) – destination node
- **request** (*Struct*) – request object (not-encoded)

Raises NodeNotReadyError – if node_id is not ready

Returns resolves to Response struct

Return type Future

set_topics(*topics*)
Set specific topics to track for metadata.

Parameters **topics** (*list of str*) – topics to check for metadata

Returns resolves after metadata request/response

Return type Future

unschedule(*task*)
Unschedule a task.

This will remove all instances of the task from the task queue. This is a no-op if the task is not scheduled.

Parameters **task** (*callable*) – task to be unscheduled

5.2.4 BrokerConnection

class kafka.**BrokerConnection**(*host*, *port*, ***configs*)

blacked_out()
Return true if we are disconnected from the given node and can't re-establish a connection yet

can_send_more()
Return True unless there are max_in_flight_requests.

close(*error=None*)
Close socket and fail all in-flight-requests.

Parameters **error** (*Exception, optional*) – pending in-flight-requests will be failed with this exception. Default: kafka.common.ConnectionError.

connect()
Attempt to connect and return ConnectionState

connected()
Return True iff socket is connected.

recv(*timeout=0*)
Non-blocking network receive.

Return response if available

send(*request*, *expect_response=True*)
send request, return Future()

Can block on network if request is larger than send_buffer_bytes

5.3 Simple APIs (DEPRECATED)

5.3.1 SimpleConsumer (DEPRECATED)

```
from kafka import SimpleProducer, SimpleClient

# To consume messages
client = SimpleClient('localhost:9092')
consumer = SimpleConsumer(client, "my-group", "my-topic")
for message in consumer:
    # message is raw byte string -- decode if necessary!
    # e.g., for unicode: `message.decode('utf-8')`
    print(message)


# Use multiprocessing for parallel consumers
from kafka import MultiProcessConsumer

# This will split the number of partitions among two processes
consumer = MultiProcessConsumer(client, "my-group", "my-topic", num_procs=2)

# This will spawn processes such that each handles 2 partitions max
consumer = MultiProcessConsumer(client, "my-group", "my-topic",
                                partitions_per_proc=2)

for message in consumer:
    print(message)

for message in consumer.get_messages(count=5, block=True, timeout=4):
    print(message)

client.close()
```

5.3.2 SimpleProducer (DEPRECATED)

Asynchronous Mode

```
from kafka import SimpleProducer, SimpleClient

# To send messages asynchronously
client = SimpleClient('localhost:9092')
producer = SimpleProducer(client, async=True)
producer.send_messages('my-topic', b'async message')

# To send messages in batch. You can use any of the available
# producers for doing this. The following producer will collect
# messages in batch and send them to Kafka after 20 messages are
# collected or every 60 seconds
# Notes:
# * If the producer dies before the messages are sent, there will be losses
# * Call producer.stop() to send the messages and cleanup
producer = SimpleProducer(client,
                          async=True,
                          batch_send_every_n=20,
                          batch_send_every_t=60)
```

Synchronous Mode

```
from kafka import SimpleProducer, SimpleClient

# To send messages synchronously
client = SimpleClient('localhost:9092')
producer = SimpleProducer(client, async=False)

# Note that the application is responsible for encoding messages to type bytes
producer.send_messages('my-topic', b'some message')
producer.send_messages('my-topic', b'this method', b'is variadic')

# Send unicode message
producer.send_messages('my-topic', u'?'.encode('utf-8'))

# To wait for acknowledgements
# ACK_AFTER_LOCAL_WRITE : server will wait till the data is written to
#                         a local log before sending response
# ACK_AFTER_CLUSTER_COMMIT : server will block until the message is committed
#                            by all in sync replicas before sending a response
producer = SimpleProducer(client,
                          async=False,
                          req_acks=SimpleProducer.ACK_AFTER_LOCAL_WRITE,
                          ack_timeout=2000,
                          sync_fail_on_error=False)

responses = producer.send_messages('my-topic', b'another message')
for r in responses:
    logging.info(r.offset)
```

5.3.3 KeyedProducer (DEPRECATED)

```
from kafka import (
    SimpleClient, KeyedProducer,
    Murmur2Partitioner, RoundRobinPartitioner)

kafka = SimpleClient('localhost:9092')

# HashedPartitioner is default (currently uses python hash())
producer = KeyedProducer(kafka)
producer.send_messages(b'my-topic', b'key1', b'some message')
producer.send_messages(b'my-topic', b'key2', b'this methode')

# Murmur2Partitioner attempts to mirror the java client hashing
producer = KeyedProducer(kafka, partitioner=Murmur2Partitioner)

# Or just produce round-robin (or just use SimpleProducer)
producer = KeyedProducer(kafka, partitioner=RoundRobinPartitioner)
```

5.3.4 SimpleClient (DEPRECATED)

```
import time
from kafka import SimpleClient
from kafka.common import (
    LeaderNotAvailableError, NotLeaderForPartitionError,
```

```
    ProduceRequestPayload)
from kafka.protocol import create_message

kafka = SimpleClient('localhost:9092')
payload = ProduceRequestPayload(topic='my-topic', partition=0,
                                messages=[create_message("some message")])

retries = 5
resps = []
while retries and not resps:
    retries -= 1
    try:
        resps = kafka.send_produce_request(
            payloads=[payload], fail_on_error=True)
    except LeaderNotAvailableError, NotLeaderForPartitionError:
        kafka.load_metadata_for_topics()
        time.sleep(1)

    # Other exceptions you might consider handling:
    # UnknownTopicOrPartitionError, TopicAuthorizationFailedError,
    # RequestTimedOutError, MessageSizeTooLargeError, InvalidTopicError,
    # RecordListTooLargeError, InvalidRequiredAcksError,
    # NotEnoughReplicasError, NotEnoughReplicasAfterAppendError

kafka.close()

resps[0].topic      # 'my-topic'
resps[0].partition  # 0
resps[0].error      # 0
resps[0].offset     # offset of the first message sent in this request
```

5.4 Install

Install with your favorite package manager

5.4.1 Latest Release

Pip:

```
pip install kafka-python
```

Releases are also listed at https://github.com/dpkp/kafka-python/releases

5.4.2 Bleeding-Edge

```
git clone https://github.com/dpkp/kafka-python
pip install ./kafka-python
```

Setuptools:

```
git clone https://github.com/dpkp/kafka-python
easy_install ./kafka-python
```

Using *setup.py* directly:

```
git clone https://github.com/dpkp/kafka-python
cd kafka-python
python setup.py install
```

5.4.3 Optional LZ4 install

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

```
>>> pip install lz4tools
>>> pip install xxhash
```

Note: these modules do not support python2.6

5.4.4 Optional Snappy install

Install Development Libraries

Download and build Snappy from http://code.google.com/p/snappy/downloads/list

Ubuntu:

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

OSX:

```
brew install snappy
```

From Source:

```
wget http://snappy.googlecode.com/files/snappy-1.0.5.tar.gz
tar xzvf snappy-1.0.5.tar.gz
cd snappy-1.0.5
./configure
make
sudo make install
```

Install Python Module

Install the *python-snappy* module

```
pip install python-snappy
```

5.5 Tests

Test environments are managed via tox. The test suite is run via pytest. Individual tests are written using unittest, pytest, and in some cases, doctest.

Linting is run via pylint, but is generally skipped on python2.6 and pypy due to pylint compatibility / performance issues.

For test coverage details, see https://coveralls.io/github/dpkp/kafka-python

The test suite includes unit tests that mock network interfaces, as well as integration tests that setup and teardown kafka broker (and zookeeper) fixtures for client / consumer / producer testing.

5.5.1 Unit tests

To run the tests locally, install tox – *pip install tox* See http://tox.readthedocs.org/en/latest/install.html

Then simply run tox, optionally setting the python environment. If unset, tox will loop through all environments.

```
tox -e py27
tox -e py35

# run protocol tests only
tox -- -v test.test_protocol

# re-run the last failing test, dropping into pdb
tox -e py27 -- --lf --pdb

# see available (pytest) options
tox -e py27 -- --help
```

5.5.2 Integration tests

```
KAFKA_VERSION=0.9.0.0 tox -e py27
KAFKA_VERSION=0.8.2.2 tox -e py35
```

Integration tests start Kafka and Zookeeper fixtures. This requires downloading kafka server binaries:

```
./build_integration.sh
```

By default, this will install 0.8.1.1, 0.8.2.2, and 0.9.0.0 brokers into the servers/ directory. To install a specific version, set *KAFKA_VERSION=1.2.3*:

```
KAFKA_VERSION=0.8.0 ./build_integration.sh
```

Then run the tests against supported Kafka versions, simply set the *KAFKA_VERSION* env variable to the server build you want to use for testing:

```
KAFKA_VERSION=0.9.0.0 tox -e py27
```

To test against the kafka source tree, set KAFKA_VERSION=trunk [optionally set SCALA_VERSION (defaults to 2.10)]

```
SCALA_VERSION=2.11 KAFKA_VERSION=trunk ./build_integration.sh
KAFKA_VERSION=trunk tox -e py35
```

5.6 Compatibility

kafka-python is compatible with (and tested against) broker versions 0.9.0.0 through 0.8.0 . kafka-python is not compatible with the 0.8.2-beta release.

kafka-python is tested on python 2.6, 2.7, 3.3, 3.4, 3.5, and pypy.

Builds and tests via Travis-CI. See https://travis-ci.org/dpkp/kafka-python

5.7 Support

For support, see github issues at https://github.com/dpkp/kafka-python

Limited IRC chat at #kafka-python on freenode (general chat is #apache-kafka).

For information about Apache Kafka generally, see https://kafka.apache.org/

For general discussion of kafka-client design and implementation (not python specific), see https://groups.google.com/forum/m/#!forum/kafka-clients

5.8 License

Apache License, v2.0. See LICENSE.

5.9 Changelog

5.9.1 1.0.0 (Feb 15, 2016)

This release includes significant code changes. Users of older kafka-python versions are encouraged to test upgrades before deploying to production as some interfaces and configuration options have changed.

Users of SimpleConsumer / SimpleProducer / SimpleClient (formerly KafkaClient) from prior releases should migrate to KafkaConsumer / KafkaProducer. Low-level APIs (Simple*) are no longer being actively maintained and will be removed in a future release.

For comprehensive API documentation, please see python help() / docstrings, kafka-python.readthedocs.org, or run 'tox -e docs' from source to build documentation locally.

Consumers

- KafkaConsumer re-written to emulate the new 0.9 kafka consumer (java client) and support coordinated consumer groups (feature requires >= 0.9.0.0 brokers)
  - Methods no longer available:
    - configure [initialize a new consumer instead]
    - set_topic_partitions [use subscribe() or assign()]
    - fetch_messages [use poll() or iterator interface]
    - get_partition_offsets
    - offsets [use committed(partition)]
    - task_done [handled internally by auto-commit; or commit offsets manually]
  - Configuration changes (consistent with updated java client):
    - lots of new configuration parameters – see docs for details
    - auto_offset_reset: previously values were 'smallest' or 'largest', now values are 'earliest' or 'latest'
    - fetch_wait_max_ms is now fetch_max_wait_ms
    - max_partition_fetch_bytes is now max_partition_fetch_bytes
    - deserializer_class is now value_deserializer and key_deserializer
    - auto_commit_enable is now enable_auto_commit

    * auto_commit_interval_messages was removed
    * socket_timeout_ms was removed
    * refresh_leader_backoff_ms was removed
- SimpleConsumer and MultiProcessConsumer are now deprecated and will be removed in a future release. Users are encouraged to migrate to KafkaConsumer.

Producers

- new producer class: KafkaProducer. Exposes the same interface as official java client. Async by default; returned future.get() can be called for synchronous blocking
- SimpleProducer is now deprecated and will be removed in a future release. Users are encouraged to migrate to KafkaProducer.

Clients

- synchronous KafkaClient renamed to SimpleClient. For backwards compatibility, you will get a SimpleClient via 'from kafka import KafkaClient'. This will change in a future release.
- All client calls use non-blocking IO under the hood.
- Add probe method check_version() to infer broker versions.

Documentation

- Updated README and sphinx documentation to address new classes.
- Docstring improvements to make python help() easier to use.

Internals

- Old protocol stack is deprecated. It has been moved to kafka.protocol.legacy and may be removed in a future release.
- Protocol layer re-written using Type classes, Schemas and Structs (modeled on the java client).
- Add support for LZ4 compression (including broken framing header checksum).

5.9.2 0.9.5 (Dec 6, 2015)

Consumers

- Initial support for consumer coordinator: offsets only (toddpalino PR 420)
- Allow blocking until some messages are received in SimpleConsumer (saaros PR 457)
- Support subclass config changes in KafkaConsumer (zackdever PR 446)
- Support retry semantics in MultiProcessConsumer (barricadeio PR 456)
- Support partition_info in MultiProcessConsumer (scrapinghub PR 418)
- Enable seek() to an absolute offset in SimpleConsumer (haosdent PR 412)
- Add KafkaConsumer.close() (ucarion PR 426)

Producers

- Catch client.reinit() exceptions in async producer (dpkp)
- Producer.stop() now blocks until async thread completes (dpkp PR 485)
- Catch errors during load_metadata_for_topics in async producer (bschopman PR 467)
- Add compression-level support for codecs that support it (trbs PR 454)
- Fix translation of Java murmur2 code, fix byte encoding for Python 3 (chrischamberlin PR 439)
- Only call stop() on not-stopped producer objects (docker-hub PR 435)
- Allow null payload for deletion feature (scrapinghub PR 409)

Clients

- Use non-blocking io for broker aware requests (ecanzonieri PR 473)
- Use debug logging level for metadata request (ecanzonieri PR 415)
- Catch KafkaUnavailableError in _send_broker_aware_request (mutability PR 436)
- Lower logging level on replica not available and commit (ecanzonieri PR 415)

Documentation

- Update docs and links wrt maintainer change (mumrah -> dpkp)

Internals

- Add py35 to tox testing
- Update travis config to use container infrastructure
- Add 0.8.2.2 and 0.9.0.0 resources for integration tests; update default official releases
- new pylint disables for pylint 1.5.1 (zackdever PR 481)
- Fix python3 / python2 comments re queue/Queue (dpkp)
- Add Murmur2Partitioner to kafka __all__ imports (dpkp Issue 471)
- Include LICENSE in PyPI sdist (koobs PR 441)

5.9.3 0.9.4 (June 11, 2015)

Consumers

- Refactor SimpleConsumer internal fetch handling (dpkp PR 399)
- Handle exceptions in SimpleConsumer commit() and reset_partition_offset() (dpkp PR 404)
- Improve FailedPayloadsError handling in KafkaConsumer (dpkp PR 398)
- KafkaConsumer: avoid raising KeyError in task_done (dpkp PR 389)
- MultiProcessConsumer – support configured partitions list (dpkp PR 380)
- Fix SimpleConsumer leadership change handling (dpkp PR 393)

- Fix SimpleConsumer connection error handling (reAsOn2010 PR 392)
- Improve Consumer handling of 'falsy' partition values (wting PR 342)
- Fix _offsets call error in KafkaConsumer (hellais PR 376)
- Fix str/bytes bug in KafkaConsumer (dpkp PR 365)
- Register atexit handlers for consumer and producer thread/multiprocess cleanup (dpkp PR 360)
- Always fetch commit offsets in base consumer unless group is None (dpkp PR 356)
- Stop consumer threads on delete (dpkp PR 357)
- Deprecate metadata_broker_list in favor of bootstrap_servers in KafkaConsumer (dpkp PR 340)
- Support pass-through parameters in multiprocess consumer (scrapinghub PR 336)
- Enable offset commit on SimpleConsumer.seek (ecanzonieri PR 350)
- Improve multiprocess consumer partition distribution (scrapinghub PR 335)
- Ignore messages with offset less than requested (wkiser PR 328)
- Handle OffsetOutOfRange in SimpleConsumer (ecanzonieri PR 296)

Producers

- Add Murmur2Partitioner (dpkp PR 378)
- Log error types in SimpleProducer and SimpleConsumer (dpkp PR 405)
- SimpleProducer support configuration of fail_on_error (dpkp PR 396)
- Deprecate KeyedProducer.send() (dpkp PR 379)
- Further improvements to async producer code (dpkp PR 388)
- Add more configuration parameters for async producer (dpkp)
- Deprecate SimpleProducer batch_send=True in favor of async (dpkp)
- Improve async producer error handling and retry logic (vshlapakov PR 331)
- Support message keys in async producer (vshlapakov PR 329)
- Use threading instead of multiprocessing for Async Producer (vshlapakov PR 330)
- Stop threads on __del__ (chmduquesne PR 324)
- Fix leadership failover handling in KeyedProducer (dpkp PR 314)

KafkaClient

- Add .topics property for list of known topics (dpkp)
- Fix request / response order guarantee bug in KafkaClient (dpkp PR 403)
- Improve KafkaClient handling of connection failures in _get_conn (dpkp)
- Client clears local metadata cache before updating from server (dpkp PR 367)
- KafkaClient should return a response or error for each request - enable better retry handling (dpkp PR 366)
- Improve str/bytes conversion in KafkaClient and KafkaConsumer (dpkp PR 332)
- Always return sorted partition ids in client.get_partition_ids_for_topic() (dpkp PR 315)

Documentation

- Cleanup Usage Documentation
- Improve KafkaConsumer documentation (dpkp PR 341)
- Update consumer documentation (sontek PR 317)
- Add doc configuration for tox (sontek PR 316)
- Switch to .rst doc format (sontek PR 321)
- Fixup google groups link in README (sontek PR 320)
- Automate documentation at kafka-python.readthedocs.org

Internals

- Switch integration testing from 0.8.2.0 to 0.8.2.1 (dpkp PR 402)
- Fix most flaky tests, improve debug logging, improve fixture handling (dpkp)
- General style cleanups (dpkp PR 394)
- Raise error on duplicate topic-partition payloads in protocol grouping (dpkp)
- Use module-level loggers instead of simply 'kafka' (dpkp)
- Remove pkg_resources check for __version__ at runtime (dpkp PR 387)
- Make external API consistently support python3 strings for topic (kecaps PR 361)
- Fix correlation id overflow (dpkp PR 355)
- Cleanup kafka/common structs (dpkp PR 338)
- Use context managers in gzip_encode / gzip_decode (dpkp PR 337)
- Save failed request as FailedPayloadsError attribute (jobevers PR 302)
- Remove unused kafka.queue (mumrah)

5.9.4 0.9.3 (Feb 3, 2015)

- Add coveralls.io support (sontek PR 307)
- Fix python2.6 threading.Event bug in ReentrantTimer (dpkp PR 312)
- Add kafka 0.8.2.0 to travis integration tests (dpkp PR 310)
- Auto-convert topics to utf-8 bytes in Producer (sontek PR 306)
- Fix reference cycle between SimpleConsumer and ReentrantTimer (zhaopengzp PR 309)
- Add Sphinx API docs (wedaly PR 282)
- Handle additional error cases exposed by 0.8.2.0 kafka server (dpkp PR 295)
- Refactor error class management (alexcb PR 289)
- Expose KafkaConsumer in __all__ for easy imports (Dinoshauer PR 286)
- SimpleProducer starts on random partition by default (alexcb PR 288)
- Add keys to compressed messages (meandthewallaby PR 281)

- Add new high-level KafkaConsumer class based on java client api (dpkp PR 234)
- Add KeyedProducer.send_messages api (pubnub PR 277)
- Fix consumer pending() method (jettify PR 276)
- Update low-level demo in README (sunisdown PR 274)
- Include key in KeyedProducer messages (se7entyse7en PR 268)
- Fix SimpleConsumer timeout behavior in get_messages (dpkp PR 238)
- Fix error in consumer.py test against max_buffer_size (rthille/wizzat PR 225/242)
- Improve string concat performance on pypy / py3 (dpkp PR 233)
- Reorg directory layout for consumer/producer/partitioners (dpkp/wizzat PR 232/243)
- Add OffsetCommitContext (locationlabs PR 217)
- Metadata Refactor (dpkp PR 223)
- Add Python 3 support (brutasse/wizzat - PR 227)
- Minor cleanups - imports / README / PyPI classifiers (dpkp - PR 221)
- Fix socket test (dpkp - PR 222)
- Fix exception catching bug in test_failover_integration (zever - PR 216)

5.9.5 0.9.2 (Aug 26, 2014)

- Warn users that async producer does not reliably handle failures (dpkp - PR 213)
- Fix spurious ConsumerFetchSizeTooSmall error in consumer (DataDog - PR 136)
- Use PyLint for static error checking (dpkp - PR 208)
- Strictly enforce str message type in producer.send_messages (dpkp - PR 211)
- Add test timers via nose-timer plugin; list 10 slowest timings by default (dpkp)
- Move fetching last known offset logic to a stand alone function (zever - PR 177)
- Improve KafkaConnection and add more tests (dpkp - PR 196)
- Raise TypeError if necessary when encoding strings (mdaniel - PR 204)
- Use Travis-CI to publish tagged releases to pypi (tkuhlman / mumrah)
- Use official binary tarballs for integration tests and parallelize travis tests (dpkp - PR 193)
- Improve new-topic creation handling (wizzat - PR 174)

5.9.6 0.9.1 (Aug 10, 2014)

- Add codec parameter to Producers to enable compression (patricklucas - PR 166)
- Support IPv6 hosts and network (snaury - PR 169)
- Remove dependency on distribute (patricklucas - PR 163)
- Fix connection error timeout and improve tests (wizzat - PR 158)
- SimpleProducer randomization of initial round robin ordering (alexcb - PR 139)
- Fix connection timeout in KafkaClient and KafkaConnection (maciejkula - PR 161)

- Fix seek + commit behavior (wizzat - PR 148)

5.9.7 0.9.0 (Mar 21, 2014)

- Connection refactor and test fixes (wizzat - PR 134)
- Fix when partition has no leader (mrtheb - PR 109)
- Change Producer API to take topic as send argument, not as instance variable (rdiomar - PR 111)
- Substantial refactor and Test Fixing (rdiomar - PR 88)
- Fix Multiprocess Consumer on windows (mahendra - PR 62)
- Improve fault tolerance; add integration tests (jimjh)
- PEP8 / Flakes / Style cleanups (Vetoshkin Nikita; mrtheb - PR 59)
- Setup Travis CI (jimjh - PR 53/54)
- Fix import of BufferUnderflowError (jimjh - PR 49)
- Fix code examples in README (StevenLeRoux - PR 47/48)

5.9.8 0.8.0

- Changing auto_commit to False in [SimpleConsumer](kafka/consumer.py), until 0.8.1 is release offset commits are unsupported
- Adding fetch_size_bytes to SimpleConsumer constructor to allow for user-configurable fetch sizes
- Allow SimpleConsumer to automatically increase the fetch size if a partial message is read and no other messages were read during that fetch request. The increase factor is 1.5
- Exception classes moved to kafka.common