Afkak

Release 20.9.0

Sep 09, 2020

Contents

Afkak is a Twisted-native Apache Kafka client library. It provides support for:

- Producing messages, with automatic batching and optional compression.
- Consuming messages, with automatic commit.

Afkak 20.9.0 was tested against:

Python CPython 2.7, 3.5+; PyPy, PyPy3 6.0+

Twisted 18.9.0

Kafka 0.9.0.1, 1.1.1

Newer broker releases will generally function, but not all Afkak features will work on older brokers. In particular, the coordinated consumer won't work before Kafka 0.9.0.1. We don't recommend deploying such old releases anyway, as they have serious bugs.

CHAPTER 1

Topics

1.1 API Documentation

1.1.1 KafkaClient objects

class afkak.**KafkaClient**(*hosts*, *clientId=None*, *timeout=10000*, *disconnect_on_timeout=False*, *correlation_id=0*, *reactor=None*, *endpoint_factory=<class 'twisted.internet.endpoints.HostnameEndpoint'>*, *retry_policy=<function backoffPolicy.<locals>.policy>*)

Cluster-aware Kafka client

KafkaClient maintains a cache of cluster metadata (brokers, topics, etc.) and routes each request to the appropriate broker connection. It must be bootstrapped with the address of at least one Kafka broker to retrieve the cluster metadata.

You will typically use this class in combination with *Producer* or *Consumer* which provide higher-level behavior.

When done with the client, call *close()* to permanently dispose of it. This terminates any open connections and release resources.

Do not set or mutate the attributes of *KafkaClient* objects. *KafkaClient* is not intended to be subclassed.

Variables

- **reactor** – Twisted reactor, as passed to the constructor. This must implement IReactorTime and IReactorTCP.
- ***clientId*** (*str*) – A short string used to identify the client to the server. This may appear in log messages on the server side.
- **_brokers** – Map of broker ID to broker metadata (host and port). This mapping is updated (mutated) whenever metadata is returned by a broker.
- **clients** – Map of broker node ID to broker clients. Items are added to this map as a connection to a specific broker is needed. Once present the client's broker metadata is updated on change.

Call `_get_brokerclient()` to get a broker client. This method constructs it and adds it to *clients* if it does not exist.

Call `_close_brokerclients()` to close a broker client once it has been removed from *clients*.

> **Warning:** Despite the name, `clients` is a private attribute.

Clients are removed when a full metadata fetch indicates that a broker no longer exists. Note that Afkak avoids doing a full metadata fetch whenever possible because it is an expensive operation, so it is possible for a broker client to remain in this map once the node is removed from the cluster. No requests will be routed to such a broker client, which will effectively leak. Afkak should be enhanced to remove such stale clients after a timeout period.

- **timeout** (*float*) – Client side request timeout, **in seconds**.

Parameters

- **timeout** (*float*) – Client-side request timeout, **in milliseconds**.
- **endpoint_factory** – Callable which accepts *reactor*, *host* and *port* arguments. It must return a `twisted.internet.interfaces.IStreamClientEndpoint`.

  Afkak does not apply a timeout to connection attempts because most endpoints include timeout logic. For example, the default of `HostnameEndpoint` applies a 30-second timeout. If an endpoint doesn't support timeouts you may need to wrap it to do so.
- **retry_policy** – Callable which accepts a count of *failures*. It returns the number of seconds (a `float`) to wait before the next attempt. This policy is used to schedule reconnection attempts to Kafka brokers.

  Use `twisted.internet.application.backoffPolicy()` to generate such a callable.

Changed in version Afkak: 3.0.0

- The *endpoint_factory* argument was added.
- The *retry_policy* argument was added.
- *timeout* may no longer be `None`. Pass a large value instead.

DEFAULT_FETCH_MIN_BYTES = 4096

DEFAULT_FETCH_SERVER_WAIT_MSECS = 5000

DEFAULT_REPLICAS_ACK_MSECS = 1000

DEFAULT_REQUEST_TIMEOUT_MSECS = 10000

clientId = 'afkak-client'

clock

close()

Permanently dispose of the client

- Immediately mark the client as closed, causing current operations to fail with *CancelledError* and future operations to fail with *ClientError*.
- Clear cached metadata.
- Close any connections to Kafka brokers.

Returns deferred that fires when all resources have been released

consumer_group_to_brokers

has_metadata_for_topic(*topic*)

load_consumer_metadata_for_group(*group*)
Deprecated alias of *load_coordinator_for_group()*

load_coordinator_for_group(*group*)
Determine the coordinator broker for the named group

Returns a deferred which callbacks with True if the group's coordinator could be determined, or errbacks with *CoordinatorNotAvailable* if not.

Parameters **group** – group name as `str`

load_metadata_for_topics(**topics*)
Discover topic metadata and brokers

Afkak internally calls this method whenever metadata is required.

Parameters **topics** (*str*) – Topic names to look up. The resulting metadata includes the list of topic partitions, brokers owning those partitions, and which partitions are in sync.

Fetching metadata for a topic may trigger auto-creation if that is enabled on the Kafka broker.

When no topic name is given metadata for *all* topics is fetched. This is an expensive operation, but it does not trigger topic creation.

Returns

`Deferred` for the completion of the metadata fetch. This will fire with `True` on success, `None` on cancellation, or fail with an exception on error.

On success, topic metadata is available from the attributes of *KafkaClient*: `topic_partitions`, `topics_to_brokers`, etc.

metadata_error_for_topic(*topic*)

partition_fully_replicated(*topic_and_part*)

reset_all_metadata()
Clear all cached metadata

Metadata will be re-fetched as required to satisfy requests.

reset_consumer_group_metadata(**groups*)
Reset cache of what broker manages the offset for specified groups

Remove the cache of what Kafka broker should be contacted when fetching or updating the committed offsets for a given consumer group or groups.

NOTE: Does not cancel any outstanding requests for updates to the consumer group metadata for the specified groups.

reset_topic_metadata(**topics*)
Remove cached metadata for the named topics

Metadata will be fetched again as required to satisfy requests.

Parameters **topics** – Topic names. Provide at least one or the method call will have no effect.

send_fetch_request(*payloads=None*, *fail_on_error=True*, *callback=None*, *max_wait_time=5000*, *min_bytes=4096*)
Encode and send a FetchRequest

Payloads are grouped by topic and partition so they can be pipelined to the same brokers.

Raises FailedPayloadsError, LeaderUnavailableError, PartitionUnavailableError

send_offset_commit_request(*group*, *payloads=None*, *fail_on_error=True*, *callback=None*, *group_generation_id=-1*, *consumer_id=''*)

Send a list of OffsetCommitRequests to the Kafka broker for the given consumer group.

Parameters

- **group** (*str*) – The consumer group to which to commit the offsets
- **payloads** (*[OffsetCommitRequest]*) – List of topic, partition, offsets to commit.
- **fail_on_error** (*bool*) – Whether to raise an exception if a response from the Kafka broker indicates an error
- **callback** (*callable*) – a function to call with each of the responses before returning the returned value to the caller.
- **group_generation_id** (*int*) – Must currently always be -1
- **consumer_id** (*str*) – Must currently always be empty string

Returns List of OffsetCommitResponse objects. Will raise KafkaError for failed requests if fail_on_error is True

Return type [*OffsetCommitResponse*]

send_offset_fetch_request(*group*, *payloads=None*, *fail_on_error=True*, *callback=None*)

Takes a group (string) and list of OffsetFetchRequest and returns a list of OffsetFetchResponse objects

send_offset_request(*payloads=None*, *fail_on_error=True*, *callback=None*)

send_produce_request(*payloads=None*, *acks=1*, *timeout=1000*, *fail_on_error=True*, *callback=None*)

Encode and send some ProduceRequests

ProduceRequests will be grouped by (topic, partition) and then sent to a specific broker. Output is a list of responses in the same order as the list of payloads specified

Parameters

- **payloads** – list of ProduceRequest
- **acks** – How many Kafka broker replicas need to write before the leader replies with a response
- **timeout** – How long the server has to receive the acks from the replicas before returning an error.
- **fail_on_error** – boolean, should we raise an Exception if we encounter an API error?
- **callback** – function, instead of returning the ProduceResponse, first pass it through this function

Returns

Return type a deferred which callbacks with a list of ProduceResponse

Raises FailedPayloadsError, LeaderUnavailableError, PartitionUnavailableError

topic_fully_replicated(*topic*)

Determine if the given topic is fully replicated according to the currently known cluster metadata.

Note: This relies on cached cluster metadata. You may call *load_metadata_for_topics()* first to refresh this cache.

Parameters **topic** (*str*) – Topic name

Returns

A boolean indicating that:

1. The number of partitions in the topic is non-zero.
2. For each partition, all replicas are in the in-sync replica (ISR) set.

Return type bool

update_cluster_hosts(*hosts*)

Advise the client of possible changes to Kafka cluster hosts

In general Afkak will keep up with changes to the cluster, but in a Docker environment where all the nodes in the cluster may change IP address at once or in quick succession Afkak may fail to track changes to the cluster.

This function lets you notify the Afkak client that some or all of the brokers may have changed. The hosts given are used the next time the client needs a fresh connection to look up cluster metadata.

Parameters **hosts** – (string|[string]) Hosts as a single comma separated "host[:port][,host[:port]]+" string, or a list of strings: ["host[:port]", ...]

1.1.2 Consumer objects

class afkak.**Consumer**(*client*, *topic*, *partition*, *processor*, *consumer_group=None*, *commit_metadata=None*, *auto_commit_every_n=None*, *auto_commit_every_ms=None*, *fetch_size_bytes=65536*, *fetch_max_wait_time=100*, *buffer_size=131072*, *max_buffer_size=None*, *request_retry_init_delay=0.1*, *request_retry_max_delay=30.0*, *request_retry_max_attempts=0*, *auto_offset_reset=None*, *commit_consumer_id=''*, *commit_generation_id=-1*)

A simple Kafka consumer implementation

This consumer consumes a single partition from a single topic, optionally automatically committing offsets. Use it as follows:

- Create an instance of *afkak.KafkaClient* with cluster connectivity details.
- Create the *Consumer*, supplying the client, topic, partition, processor function, and optionally fetch specifics, a consumer group, and a commit policy.
- Call *start()* with the offset within the partition at which to start consuming messages. See *start()* for details.
- Process the messages in your `processor` callback, returning a `Deferred` to provide backpressure as needed.
- Once processing resolves, `processor` will be called again with the next batch of messages.
- When desired, call *shutdown()* on the *Consumer* to halt calls to the `processor` function and commit progress (if a *consumer_group* is specified).

A *Consumer* may be restarted once stopped.

Variables

- **client** – Connected *KafkaClient* for submitting requests to the Kafka cluster.
- **topic** (*str*) – The topic from which to consume messages.
- **partition** (*int*) – The partition from which to consume.
- **processor** (*callable*) – The callback function to which the consumer and lists of messages (*afkak.common.SourcedMessage*) will be submitted for processing. The function may return a `Deferred` and will not be called again until this Deferred resolves.
- **consumer_group** (*str*) – Optional consumer group ID for committing offsets of processed messages back to Kafka.
- **commit_metadata** (*bytes*) – Optional metadata to store with offsets commit.
- **auto_commit_every_n** (*int*) – Number of messages after which the consumer will automatically commit the offset of the last processed message to Kafka. Zero disables, defaulted to `AUTO_COMMIT_MSG_COUNT`.
- **auto_commit_every_ms** (*int*) – Time interval in milliseconds after which the consumer will automatically commit the offset of the last processed message to Kafka. Zero disables, defaulted to `AUTO_COMMIT_INTERVAL`.
- **fetch_size_bytes** (*int*) – Number of bytes to request in a `FetchRequest`. Kafka will defer fulfilling the request until at least this many bytes can be returned.
- **fetch_max_wait_time** (*int*) – Max number of milliseconds the server should wait for that many bytes.
- **buffer_size** (*int*) – default 128K. Initial number of bytes to tell Kafka we have available. This will be raised x16 up to 1MB then double up to...
- **max_buffer_size** (*int*) – Max number of bytes to tell Kafka we have available. `None` means no limit (the default). Must be larger than the largest message we will find in our topic/partitions.
- **request_retry_init_delay** (*float*) – Number of seconds to wait before retrying a failed request to Kafka.
- **request_retry_max_delay** (*float*) – Maximum number of seconds to wait before retrying a failed request to Kafka (the delay is increased on each failure and reset to the initial delay upon success).
- **request_retry_max_attempts** (*int*) – Maximum number of attempts to make for any request. Default of zero means retry forever; other values must be positive and indicate the number of attempts to make before returning failure.
- **auto_offset_reset** (*int*) – What action should be taken when the broker responds to a fetch request with *OffsetOutOfRangeError*?
  - *OFFSET_EARLIEST*: request the oldest available messages. The consumer will read every message in the topic.
  - *OFFSET_LATEST*: request the most recent messages (this is the Java consumer's default). The consumer will read messages once new messages are produced to the topic.
  - `None`: fail on *OffsetOutOfRangeError* (Afkak's default). The `Deferred` returned by *Consumer.start()* will errback. The caller may call *start()* again with the desired offset.

The broker returns *OffsetOutOfRangeError* when the client requests an offset that isn't valid. This may mean that the requested offset no longer exists, e.g. if it was removed due to age.

commit()
Commit the last processed offset

Immediately commit the value of *last_processed_offset* if it differs from *last_committed_offset*.

Note: It is possible to commit a smaller offset than Kafka has stored. This is by design, so we can reprocess a Kafka message stream if desired.

On error, will retry according to `request_retry_max_attempts` (by default, forever).

If called while a commit operation is in progress, and new messages have been processed since the last request was sent then the commit will fail with `OperationInProgress`. The `OperationInProgress` exception wraps a `Deferred` which fires when the outstanding commit operation completes.

Returns A `Deferred` which resolves with the committed offset when the operation has completed. It will resolve immediately if the current offset and the last committed offset do not differ.

last_committed_offset
The last offset that was successfully commited to Kafka, or `None` if no offset has been committed yet (read-only).

Return type Optional[int]

last_processed_offset
Offset of the last message that was successfully processed, or `None` if no message has been processed yet (read-only). This is updated only once the processor function returns and any deferred it returns succeeds.

Return type Optional[int]

shutdown()
Gracefully shutdown the consumer

Consumer will complete any outstanding processing, commit its current offsets (if so configured) and stop.

Returns `Deferred` that fires with the value of *last_processed_offset*. It may fail if a commit fails or with `RestopError` if the consumer is not running.

start(*start_offset*)
Starts fetching messages from Kafka and delivering them to the `processor` function.

Parameters **start_offset** (*int*) – The offset within the partition from which to start fetching. Special values include: `OFFSET_EARLIEST`, `OFFSET_LATEST`, and `OFFSET_COMMITTED`. If the supplied offset is `OFFSET_EARLIEST` or `OFFSET_LATEST` the *Consumer* will use the OffsetRequest Kafka API to retrieve the actual offset used for fetching. In the case `OFFSET_COMMITTED` is used, `commit_policy` MUST be set on the Consumer, and the Consumer will use the OffsetFetchRequest Kafka API to retrieve the actual offset used for fetching.

Returns

`Deferred` that will fire when the consumer is stopped:

- It will succeed with the value of *last_processed_offset*, or

- Fail when the *Consumer* encounters an error from which it is unable to recover, such as an exception thrown by the processor or an unretriable broker error.

Raises RestartError if already running.

stop()

Stop the consumer and return offset of last processed message. This cancels all outstanding operations. Also, if the deferred returned by *start* hasn't been called, it is called with the value of *last_processed_offset*.

Raises RestopError if the *Consumer* is not running.

1.1.3 Offset constants

afkak.**OFFSET_EARLIEST**

afkak.**OFFSET_LATEST**

afkak.**OFFSET_COMMITTED**

1.1.4 Producer objects

class afkak.**Producer**(*client*, *partitioner_class=<class 'afkak.partitioner.RoundRobinPartitioner'>*, *req_acks=1*, *ack_timeout=1000*, *max_req_attempts=10*, *retry_interval=0.25*, *codec=None*, *batch_send=False*, *batch_every_n=10*, *batch_every_b=32768*, *batch_every_t=30*)

Write messages to Kafka with retries and batching

Parameters

- **client** – *KafkaClient* instance to use
- **partitioner_class** – Factory for topic partitioners, a callable that accepts a topic and list of partition numbers. The default is RoundRobinPartitioner.
- **req_acks** (*int*) – A value indicating the acknowledgements that the server must receive before responding to the request
- **ack_timeout** (*float*) – Value (in milliseconds) indicating a how long the server can wait for the above acknowledgements.
- **max_req_attempts** (*int*) – Number of times we will retry a request to Kafka before failing the request.
- **retry_interval** (*float*) – Initial retry interval in seconds, defaults to INIT_RETRY_INTERVAL.
- **codec** – Compression codec to apply to messages. Default: *CODEC_NONE*.
- **batch_send** (*bool*) – If True, messages are sent in batches.
- **batch_every_n** (*int*) – If set, messages are sent in batches of this many messages.
- **batch_every_b** (*int*) – If set, messages are sent when this many bytes of messages are waiting to be sent.
- **batch_every_t** (*int*) – If set, messages are sent after this many seconds (even if waiting for other conditions to apply). This caps the latency automatic batching incurs.

send_messages(*topic*, *key=None*, *msgs=()*)
Given a topic, and optional key (for partitioning) and a list of messages, send them to Kafka, either immediately, or when a batch is ready, depending on the Producer's batch settings.

Parameters

- **topic** (*str*) – Kafka topic to send the messages to
- **key** (*str*) – Message key used to determine the topic partition to which the messages will be written. Either `bytes` or `None`.

  `None` means that there is no key, but note that that:

  – Kafka does not permit producing unkeyed messages to a compacted topic.

  – The *partitioner_class* may require a non-`None` key (`HashedPartitioner` does so).
- **msgs** (*list*) – A non-empty sequence of message bytestrings to send. `None` indicates a `null` message (i.e. a tombstone on a compacted topic).

Returns

A `Deferred` that fires when the messages have been received by the Kafka cluster.

It will fail with `TypeError` when:

- *topic* is not text (`str` on Python 3, `str` or `unicode` on Python 2)
- *key* is not `bytes` or `None`
- *msgs* is not a sequence of `bytes` or `None`

It will fail with `ValueError` when *msgs* is empty.

stop()
Terminate any outstanding requests.

Returns :class:‘‘Deferred‘ which fires when fully stopped.

1.1.5 Compression constants

afkak.**CODEC_NONE**
No compression.

afkak.**CODEC_GZIP**
Gzip compression.

afkak.**CODEC_SNAPPY**
Snappy compression.

Snappy compression requires Afkak's `snappy` extra. For example:

```
pip install afkak[snappy]
```

afkak.**CODEC_LZ4**
LZ4 compression. Not currently supported by Afkak.

1.1.6 Partitioners

1.1.7 Message construction

Use these functions to construct payloads to send with *KafkaClient.send_produce_request()*.

afkak.**create_message**(*payload*, *key=None*)
Construct a Message

Parameters

- **payload** (bytes or None) – The payload to send to Kafka.
- **key** (bytes or None) – A key used to route the message when partitioning and to determine message identity on a compacted topic.

afkak.**create_message_set**(*requests*, *codec=0*)
Create a message set from a list of requests.

Each request can have a list of messages and its own key. If codec is CODEC_NONE, return a list of raw Kafka messages. Otherwise, return a list containing a single codec-encoded message.

Parameters **codec** – The encoding for the message set, one of the constants:

- *afkak.CODEC_NONE*
- *afkak.CODEC_GZIP*
- *afkak.CODEC_SNAPPY*

Raises UnsupportedCodecError for an unsupported codec

1.1.8 Common objects

class afkak.common.**BrokerMetadata**(*node_id*, *host*, *port*)

host
Alias for field number 1

node_id
Alias for field number 0

port
Alias for field number 2

exception afkak.common.**BrokerNotAvailableError**

errno = 8

message = 'BROKER_NOT_AVAILABLE'

exception afkak.common.**BrokerResponseError**
One *BrokerResponseError* subclass is defined for each protocol error code.

Variables

- ***errno*** (*int*) – The integer error code reported by the server.
- ***retriable*** (*bool*) – A flag which indicates whether it is valid to retry the request which produced the error. Note that a metadata refresh may be required before retry, depending on the type of error.
- ***message*** (*str*) – The error code string, per the table. None if the error code is unknown to Afkak (future Kafka releases may add additional error codes). Note that this value may change for a given exception type. Code should either check the exception type or errno.

errnos = {-1: <class 'afkak.common.UnknownError'>, 1: <class 'afkak.common.OffsetOut

message = None

classmethod raise_for_errno(*errno*, **args*)
Raise an exception for the given error number.

Parameters **errno** (*int*) – Kafka error code.

Raises ***BrokerResponseError*** – For any non-zero *errno* a *BrokerResponseError* is raised. If Afkak defines a specific exception type for the error code that is raised. All such types subclass *BrokerResponseError*.

retriable = False

exception afkak.common.**BufferUnderflowError**

exception afkak.common.**CancelledError**(*request_sent=None*, *message=None*)

exception afkak.common.**ChecksumError**

exception afkak.common.**ClientError**
Generic error when the client detects an error

exception afkak.common.**ClusterAuthorizationFailed**

errno = 31

message = 'CLUSTER_AUTHORIZATION_FAILED'

exception afkak.common.**ConcurrentTransactions**

errno = 51

message = 'CONCURRENT_TRANSACTIONS'

exception afkak.common.**ConnectionError**

afkak.common.**ConsumerCoordinatorNotAvailableError**
alias of *afkak.common.CoordinatorNotAvailable*

exception afkak.common.**ConsumerFetchSizeTooSmall**

class afkak.common.**ConsumerMetadataResponse**(*error*, *node_id*, *host*, *port*)

error
Alias for field number 0

host
Alias for field number 2

node_id
Alias for field number 1

port
Alias for field number 3

exception afkak.common.**CoordinatorLoadInProgress**

errno = 14

message = 'COORDINATOR_LOAD_IN_PROGRESS'

exception afkak.common.**CoordinatorNotAvailable**

errno = 15

```
    message = 'COORDINATOR_NOT_AVAILABLE'
exception afkak.common.CorruptMessage

    errno = 2
    message = 'CORRUPT_MESSAGE'
exception afkak.common.DelegationTokenAuthDisabled

    errno = 61
    message = 'DELEGATION_TOKEN_AUTH_DISABLED'
exception afkak.common.DelegationTokenAuthorizationFailed

    errno = 65
    message = 'DELEGATION_TOKEN_AUTHORIZATION_FAILED'
exception afkak.common.DelegationTokenExpired

    errno = 66
    message = 'DELEGATION_TOKEN_EXPIRED'
exception afkak.common.DelegationTokenNotFound

    errno = 62
    message = 'DELEGATION_TOKEN_NOT_FOUND'
exception afkak.common.DelegationTokenOwnerMismatch

    errno = 63
    message = 'DELEGATION_TOKEN_OWNER_MISMATCH'
exception afkak.common.DelegationTokenRequestNotAllowed

    errno = 64
    message = 'DELEGATION_TOKEN_REQUEST_NOT_ALLOWED'
exception afkak.common.DuplicateRequestError
```

Error caused by calling makeRequest() with a duplicate requestId

```
exception afkak.common.DuplicateSequenceNumber

    errno = 46
    message = 'DUPLICATE_SEQUENCE_NUMBER'
exception afkak.common.FailedPayloadsError
```

FailedPayloadsError indicates a partial or total failure

In a method like *KafkaClient.send_produce_request()* partial failure is possible because payloads are distributed among the Kafka brokers that lead each partition.

Variables

- ***responses*** (*list*) – Any successful responses.
- ***failed_payloads*** (*list*) – Two-tuples of (payload, failure).

failed_payloads

responses

class afkak.common.**FetchRequest**(*topic*, *partition*, *offset*, *max_bytes*)

max_bytes
Alias for field number 3

offset
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

class afkak.common.**FetchResponse**(*topic*, *partition*, *error*, *highwaterMark*, *messages*)

error
Alias for field number 2

highwaterMark
Alias for field number 3

messages
Alias for field number 4

partition
Alias for field number 1

topic
Alias for field number 0

exception afkak.common.**FetchSessionIdNotFound**

errno = 70

message = 'FETCH_SESSION_ID_NOT_FOUND'

exception afkak.common.**GroupAuthorizationFailed**

errno = 30

message = 'GROUP_AUTHORIZATION_FAILED'

exception afkak.common.**GroupIdNotFound**

errno = 69

message = 'GROUP_ID_NOT_FOUND'

exception afkak.common.**IllegalGeneration**

errno = 22

```
    message = 'ILLEGAL_GENERATION'
exception afkak.common.IllegalSaslState

    errno = 34
    message = 'ILLEGAL_SASL_STATE'
exception afkak.common.InconsistentGroupProtocol

    errno = 23
    message = 'INCONSISTENT_GROUP_PROTOCOL'
exception afkak.common.InvalidCommitOffsetSize

    errno = 28
    message = 'INVALID_COMMIT_OFFSET_SIZE'
exception afkak.common.InvalidConfig

    errno = 40
    message = 'INVALID_CONFIG'
exception afkak.common.InvalidConsumerGroupError
exception afkak.common.InvalidFetchRequestError

    errno = 4
    message = 'INVALID_FETCH_SIZE'
exception afkak.common.InvalidFetchSessionEpoch

    errno = 71
    message = 'INVALID_FETCH_SESSION_EPOCH'
exception afkak.common.InvalidGroupId

    errno = 24
    message = 'INVALID_GROUP_ID'
afkak.common.InvalidMessageError
    alias of afkak.common.CorruptMessage
exception afkak.common.InvalidPartitions

    errno = 37
    message = 'INVALID_PARTITIONS'
exception afkak.common.InvalidPrincipalType

    errno = 67
```

```
    message = 'INVALID_PRINCIPAL_TYPE'
exception afkak.common.InvalidProducerEpoch

    errno = 47
    message = 'INVALID_PRODUCER_EPOCH'
exception afkak.common.InvalidProducerIdMapping

    errno = 49
    message = 'INVALID_PRODUCER_ID_MAPPING'
exception afkak.common.InvalidReplicaAssignment

    errno = 39
    message = 'INVALID_REPLICA_ASSIGNMENT'
exception afkak.common.InvalidReplicationFactor

    errno = 38
    message = 'INVALID_REPLICATION_FACTOR'
exception afkak.common.InvalidRequest

    errno = 42
    message = 'INVALID_REQUEST'
exception afkak.common.InvalidRequiredAcks

    errno = 21
    message = 'INVALID_REQUIRED_ACKS'
exception afkak.common.InvalidSessionTimeout

    errno = 26
    message = 'INVALID_SESSION_TIMEOUT'
exception afkak.common.InvalidTimestamp

    errno = 32
    message = 'INVALID_TIMESTAMP'
exception afkak.common.InvalidTopic
```

The request specified an illegal topic name. The name is either malformed or references an internal topic for which the operation is not valid.

```
    errno = 17
    message = 'INVALID_TOPIC_EXCEPTION'
exception afkak.common.InvalidTransactionTimeout
```

errno = 50

message = 'INVALID_TRANSACTION_TIMEOUT'

exception afkak.common.**InvalidTxnState**

errno = 48

message = 'INVALID_TXN_STATE'

exception afkak.common.**KafkaError**

exception afkak.common.**KafkaStorageError**

errno = 56

message = 'KAFKA_STORAGE_ERROR'

exception afkak.common.**KafkaUnavailableError**

exception afkak.common.**LeaderNotAvailableError**

errno = 5

message = 'LEADER_NOT_AVAILABLE'

exception afkak.common.**LeaderUnavailableError**

exception afkak.common.**ListenerNotFound**

errno = 72

message = 'LISTENER_NOT_FOUND'

exception afkak.common.**LogDirNotFound**

errno = 57

message = 'LOG_DIR_NOT_FOUND'

class afkak.common.**Message**

A Kafka message in format 0.

Variables

- **magic** (*int*) – Message format version, always 0.
- **attributes** (*int*) – Compression flags.
- ***key*** (*bytes*) – Message key, or `None` when the message lacks a key. Note that the key is required on a compacted topic.
- **value** (*bytes*) – Message value, or `None` if this is a tombstone a.k.a. null message.

exception afkak.common.**MessageSizeTooLargeError**

errno = 10

message = 'MESSAGE_SIZE_TOO_LARGE'

exception afkak.common.**NetworkException**

errno = 13

message = 'NETWORK_EXCEPTION'

exception afkak.common.**NoResponseError**

exception afkak.common.**NonEmptyGroup**

errno = 68

message = 'NON_EMPTY_GROUP'

exception afkak.common.**NotController**

errno = 41

message = 'NOT_CONTROLLER'

exception afkak.common.**NotCoordinator**

errno = 16

message = 'NOT_COORDINATOR'

afkak.common.**NotCoordinatorForConsumerError**
alias of *afkak.common.NotCoordinator*

exception afkak.common.**NotEnoughReplicas**
The number of in-sync replicas is lower than can satisfy the number of acks required by the produce request.

errno = 19

message = 'NOT_ENOUGH_REPLICAS'

exception afkak.common.**NotEnoughReplicasAfterAppend**
The produce request was written to the log, but not by as many in-sync replicas as it required.

errno = 20

message = 'NOT_ENOUGH_REPLICAS_AFTER_APPEND'

exception afkak.common.**NotLeaderForPartitionError**

errno = 6

message = 'NOT_LEADER_FOR_PARTITION'

class afkak.common.**OffsetAndMessage**(*offset*, *message*)

message
Alias for field number 1

offset
Alias for field number 0

class afkak.common.**OffsetCommitRequest**(*topic*, *partition*, *offset*, *timestamp*, *metadata*)

metadata
Alias for field number 4

offset
Alias for field number 2

partition
Alias for field number 1

timestamp
Alias for field number 3

topic
Alias for field number 0

class afkak.common.**OffsetCommitResponse**(*topic*, *partition*, *error*)

error
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

class afkak.common.**OffsetFetchRequest**(*topic*, *partition*)

partition
Alias for field number 1

topic
Alias for field number 0

class afkak.common.**OffsetFetchResponse**(*topic*, *partition*, *offset*, *metadata*, *error*)

error
Alias for field number 4

metadata
Alias for field number 3

offset
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

exception afkak.common.**OffsetMetadataTooLargeError**

errno = 12

message = 'OFFSET_METADATA_TOO_LARGE'

exception afkak.common.**OffsetOutOfRangeError**

errno = 1

message = 'OFFSET_OUT_OF_RANGE'

class afkak.common.**OffsetRequest**(*topic*, *partition*, *time*, *max_offsets*)

max_offsets
Alias for field number 3

partition
Alias for field number 1

time
Alias for field number 2

topic
Alias for field number 0

class afkak.common.**OffsetResponse**(*topic*, *partition*, *error*, *offsets*)

error
Alias for field number 2

offsets
Alias for field number 3

partition
Alias for field number 1

topic
Alias for field number 0

afkak.common.**OffsetsLoadInProgressError**
alias of *afkak.common.CoordinatorLoadInProgress*

exception afkak.common.**OperationInProgress**(*deferred=None*)

exception afkak.common.**OperationNotAttempted**

errno = 55

message = 'OPERATION_NOT_ATTEMPTED'

exception afkak.common.**OutOfOrderSequenceNumber**

errno = 45

message = 'OUT_OF_ORDER_SEQUENCE_NUMBER'

class afkak.common.**PartitionMetadata**(*topic*, *partition*, *partition_error_code*, *leader*, *replicas*, *isr*)

isr
Alias for field number 5

leader
Alias for field number 3

partition
Alias for field number 1

partition_error_code
Alias for field number 2

replicas
Alias for field number 4

topic
Alias for field number 0

exception afkak.common.**PartitionUnavailableError**

exception afkak.common.**PolicyViolation**

errno = 44

message = 'POLICY_VIOLATION'

class afkak.common.**ProduceRequest**(*topic*, *partition*, *messages*)

messages
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

class afkak.common.**ProduceResponse**(*topic*, *partition*, *error*, *offset*)

error
Alias for field number 2

offset
Alias for field number 3

partition
Alias for field number 1

topic
Alias for field number 0

exception afkak.common.**ProtocolError**

exception afkak.common.**ReassignmentInProgress**

errno = 60

message = 'REASSIGNMENT_IN_PROGRESS'

exception afkak.common.**RebalanceInProgress**

errno = 27

message = 'REBALANCE_IN_PROGRESS'

exception afkak.common.**RecordListTooLarge**
The produce request message batch exceeds the maximum configured segment size.

errno = 18

message = 'RECORD_LIST_TOO_LARGE'

exception afkak.common.**ReplicaNotAvailableError**

errno = 9

message = 'REPLICA_NOT_AVAILABLE'

exception afkak.common.**RequestTimedOutError**

errno = 7

message = 'REQUEST_TIMED_OUT'

exception afkak.common.**RestartError**
Raised when a consumer start() call is made on an already running consumer

exception afkak.common.**RestopError**
Raised when a consumer stop() or shutdown() call is made on a non-running consumer

exception afkak.common.**RetriableBrokerResponseError**
RetriableBrokerResponseError is the shared superclass of all broker errors which can be retried.

retriable = True

exception afkak.common.**SaslAuthenticationFailed**

errno = 58

message = 'SASL_AUTHENTICATION_FAILED'

exception afkak.common.**SecurityDisabled**

errno = 54

message = 'SECURITY_DISABLED'

class afkak.common.**SendRequest**(*topic*, *key*, *messages*, *deferred*)

deferred
Alias for field number 3

key
Alias for field number 1

messages
Alias for field number 2

topic
Alias for field number 0

class afkak.common.**SourcedMessage**(*topic*, *partition*, *offset*, *message*)

message
Alias for field number 3

offset
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

exception afkak.common.**StaleControllerEpochError**

errno = 11

message = 'STALE_CONTROLLER_EPOCH'

afkak.common.**StaleLeaderEpochCodeError**
alias of *afkak.common.NetworkException*

exception afkak.common.**TopicAlreadyExists**

errno = 36

message = 'TOPIC_ALREADY_EXISTS'

class afkak.common.**TopicAndPartition**(*topic*, *partition*)

partition
Alias for field number 1

topic
Alias for field number 0

exception afkak.common.**TopicAuthorizationFailed**

errno = 29

message = 'TOPIC_AUTHORIZATION_FAILED'

class afkak.common.**TopicMetadata**(*topic*, *topic_error_code*, *partition_metadata*)

partition_metadata
Alias for field number 2

topic
Alias for field number 0

topic_error_code
Alias for field number 1

exception afkak.common.**TransactionCoordinatorFenced**

errno = 52

message = 'TRANSACTION_COORDINATOR_FENCED'

exception afkak.common.**TransactionalIdAuthorizationFailed**

errno = 53

message = 'TRANSACTIONAL_ID_AUTHORIZATION_FAILED'

exception afkak.common.**UnknownError**

errno = -1

```
    message = 'UNKNOWN_SERVER_ERROR'
exception afkak.common.UnknownMemberId

    errno = 25
    message = 'UNKNOWN_MEMBER_ID'
exception afkak.common.UnknownProducerId

    errno = 59
    message = 'UNKNOWN_PRODUCER_ID'
exception afkak.common.UnknownTopicOrPartitionError

    errno = 3
    message = 'UNKNOWN_TOPIC_OR_PARTITION'
exception afkak.common.UnsupportedCodecError
exception afkak.common.UnsupportedForMessageFormat

    errno = 43
    message = 'UNSUPPORTED_FOR_MESSAGE_FORMAT'
exception afkak.common.UnsupportedSaslMechanism

    errno = 33
    message = 'UNSUPPORTED_SASL_MECHANISM'
exception afkak.common.UnsupportedVersion

    errno = 35
    message = 'UNSUPPORTED_VERSION'
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

F

G

H

I

K

L

M

N

O

P

R

S

T

U