pika Documentation

Release 0.9.6

Gavin M. Roy

December 14, 2016

Contents

Pika is a pure-Python implementation of the AMQP 0-9-1 protocol that tries to stay fairly independent of the underlying network support library.

This documentation is a combination of both user documentation and module development documentation. Modules and classes called out in the Using Pika section below will cover a majority of what users who are implementation pika in their applications will need. The Pika Core Objects section below lists all of the modules that are internal to Pika.

If you have not developed with Pika or RabbitMQ before, the Connecting to RabbitMQ documentation is a good place to get started.

CHAPTER 1

Installing Pika

Pika is available for download via PyPI and may be installed using easy_install or pip:

```
pip install pika
```

or:

```
easy_install pika
```

To install from source, run "python setup.py install" in the root source directory.

CHAPTER 2

Using Pika

Pika supports two modes of development, synchronous using the BlockingConnection adapter and asynchronous using one of the AsyncoreConnection, SelectConnection and TwistedConnection adapters.

2.1 Connecting to RabbitMQ

Pika provides multiple adapters to connect to RabbitMQ allowing for different ways of providing socket communication depending on what is appropriate for your application.

- *SelectConnection*: A native event based connection adapter that implements select, kqueue, poll and epoll.
- *AsyncoreConnection*: Legacy adapter kept for convenience of previous Pika users.
- *TornadoConnection*: Connection adapter for use with the Tornado IO Loop.
- *BlockingConnection*: Enables blocking, synchronous operation on top of library for simple uses.

2.1.1 IO and Event Looping

Due to the need to check for and send content on a consistent basis, Pika now implements or extends IOLoops in each of its asynchronous connection adapters. These IOLoops are blocking methods which loop and listen for events. Each asynchronous adapters follows the same standard for invoking the IOLoop. The IOLoop is created when the connection adapter is created. To start it simply call the connection.ioloop.start() method.

If you are using an external IOLoop such as Tornado's IOLoop, you may invoke that as you normally would and then add the adapter to it.

Example:

```
from pika.adapters import SelectConnection

# Create our connection object
connection = SelectConnection()

try:
    # Loop so we can communicate with RabbitMQ
    connection.ioloop.start()
except KeyboardInterrupt:
    # Gracefully close the connection
    connection.close()
    # Loop until we're fully closed, will stop on its own
    connection.ioloop.start()
```

2.1.2 Continuation-Passing Style

Interfacing with Pika asynchronously is done by passing in callback methods you would like to have invoked when a certain event has completed. For example, if you are going to declare a queue, you pass in a method that will be called when the RabbitMQ server returns a Queue.DeclareOk response.

In our example below we use the following four easy steps:

1. We start by creating our connection object, then starting our event loop.
2. When we are connected, the *on_connected* method is called. In that method we create a channel.
3. When the channel is created, the *on_channel_open* method is called. In that method we declare a queue.
4. When the queue is declared successfully, *on_queue_declared* is called. In that method we call `channel.basic_consume` telling it to call the handle_delivery for each message RabbitMQ delivers to us.
5. When RabbitMQ has a message to send us, it call the handle_delivery method passing the AMQP Method frame, Header frame and Body.

Note: Step #1 is on line #28 and Step #2 is on line #6. This is so that Python knows about the functions we'll call in Steps #2 through #5.

Example:

```
import pika

# Create a global channel variable to hold our channel object in
channel = None

# Step #2
def on_connected(connection):
    """Called when we are fully connected to RabbitMQ"""
    # Open a channel
    connection.channel(on_channel_open)

# Step #3
def on_channel_open(new_channel):
    """Called when our channel has opened"""
    global channel
    channel = new_channel
    channel.queue_declare(queue="test", durable=True, exclusive=False, auto_delete=False, callback=on

# Step #4
def on_queue_declared(frame):
    """Called when RabbitMQ has told us our Queue has been declared, frame is the response from Rabbi
    channel.basic_consume(handle_delivery, queue='test')

# Step #5
def handle_delivery(channel, method, header, body):
    """Called when we receive a message from RabbitMQ"""
    print body

# Step #1: Connect to RabbitMQ using the default parameters
parameters = pika.ConnectionParameters()
connection = pika.SelectConnection(parameters, on_connected)

try:
```

```
    # Loop so we can communicate with RabbitMQ
    connection.ioloop.start()
except KeyboardInterrupt:
    # Gracefully close the connection
    connection.close()
    # Loop until we're fully closed, will stop on its own
    connection.ioloop.start()
```

2.1.3 Credentials

The credentials module provides the mechanism by which you pass the username and password to the `connection.ConnectionParameters()` class when it is created.

class `pika.credentials.`**`PlainCredentials`**(*username*, *password*, *erase_on_connect=False*)

The PlainCredentials class returns the properly formatted username and password to the Connection. As of this version of Pika, only PlainCredentials are supported. To authenticate with Pika, simply create a credentials object passing in the username and password and pass that to the ConnectionParameters object.

If you do not pass in credentials to the ConnectionParameters object, it will create credentials for 'guest' with the password of 'guest'.

If you pass True to erase_on_connect the credentials will not be stored in memory after the Connection attempt has been made.

`response_for`(*start*)

Validate that this type of authentication is supported

Parameters **`start`** (`spec.Connection.Start`) – Connection.Start method

Return type tuple(str|None, str|None)

`erase_credentials`()

Called by Connection when it no longer needs the credentials

Example:

```
import pika
credentials = pika.PlainCredentials('username', 'password')
parameters = pika.ConnectionParameters(credentials=credentials)
```

2.1.4 Connection Parameters

There are two types of connection parameter classes in Pika to allow you to pass the connection information into a connection adapter, ConnectionParameters and URLParameters. Both classes share the same default connection values.

Default Parameter Values

The connection parameters classes extend pika.connection.Parameters to create a consistent definition of default values and internal attributes.

`pika.connection.`**`Parameters`** **= <class 'pika.connection.Parameters'>**

Base connection parameters class definition

Parameters

- **`DEFAULT_HOST`** (*str*) – 'localhost'

- **DEFAULT_PORT** (*int*) – 5672
- **DEFAULT_VIRTUAL_HOST** (*str*) – '/'
- **DEFAULT_USERNAME** (*str*) – 'guest'
- **DEFAULT_PASSWORD** (*str*) – 'guest'
- **DEFAULT_HEARTBEAT_INTERVAL** (*int*) – 0
- **DEFAULT_CHANNEL_MAX** (*int*) – 0
- **DEFAULT_FRAME_MAX** (*int*) – pika.spec.FRAME_MAX_SIZE
- **DEFAULT_LOCALE** (*str*) – 'en_US'
- **DEFAULT_CONNECTION_ATTEMPTS** (*int*) – 1
- **DEFAULT_RETRY_DELAY** (*int | float*) – 2.0
- **DEFAULT_SOCKET_TIMEOUT** (*int | float*) – 0.25
- **DEFAULT_SSL** (*bool*) – False
- **DEFAULT_SSL_OPTIONS** (*dict*) – {}
- **DEFAULT_SSL_PORT** (*int*) – 5671
- **DEFAULT_BACKPRESSURE_DETECTION** (*bool*) – False

ConnectionParameters

The ConnectionParameters class allows you to specify the options needed when creating the object.

class pika.connection.**ConnectionParameters**(*host=None*, *port=None*, *virtual_host=None*, *credentials=None*, *channel_max=None*, *frame_max=None*, *heartbeat_interval=None*, *ssl=None*, *ssl_options=None*, *connection_attempts=None*, *retry_delay=None*, *socket_timeout=None*, *locale=None*, *backpressure_detection=None*)

Connection parameters object that is passed into the connection adapter upon construction.

Example:

```
import pika

# Set the connection parameters to connect to rabbit-server1 on port 5672
# on the / virtual host using the username "guest" and password "guest"
credentials = pika.PlainCredentials('guest', 'guest')
parameters = pika.ConnectionParameters('rabbit-server1',
                                       5672
                                       '/',
                                       credentials)
```

URLParameters

The URLParameters class allows you to pass in an AMQP URL when creating the object and supports the host, port, virtual host, ssl, username and password in the base URL and other options are passed in via query parameters.

class pika.connection.**URLParameters**(*url*)

Create a Connection parameters object based off of URIParameters

Example:

```
import pika

# Set the connection parameters to connect to rabbit-server1 on port 5672
# on the / virtual host using the username "guest" and password "guest"
parameters = pika.URLParameters('amqp://guest:guest@rabbit-server1:5672/%2F')
```

2.1.5 TCP Backpressure

As of RabbitMQ 2.0, client side Channel.Flow has been removed [1]. Instead, the RabbitMQ broker uses TCP Backpressure to slow your client if it is delivering messages too fast. If you pass in backpressure_detection into your connection parameters, Pika attempts to help you handle this situation by providing a mechanism by which you may be notified if Pika has noticed too many frames have yet to be delivered. By registering a callback function with the `add_backpressure_callback` method of any connection adapter, your function will be called when Pika sees that a backlog of 10 times the average frame size you have been sending has been exceeded. You may tweak the notification multiplier value by calling the `set_backpressure_multiplier` method passing any integer value.

Example:

```
import pika

parameters = pika.URLParameters('amqp://guest:guest@rabbit-server1:5672/%2F?backpressure_detection=t
```

2.1.6 Available Adapters

The following connection adapters are available for connecting with RabbitMQ:

AsyncoreConnection

Note: Use It is recommended that you use SelectConnection and its method signatures are the same as AsyncoreConnection.

The AsyncoreConnection class is provided for legacy support and quicker porting from applications that used Pika version 0.5.2 and prior.

class `pika.adapters.asyncore_connection.`**`AsyncoreConnection`**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=True*)

`add_backpressure_callback`(*callback_method*)
Call method "callback" when pika believes backpressure is being applied.

Parameters **`callback_method`** (*method*) – The method to call

`add_on_close_callback`(*callback_method*)
Add a callback notification when the connection has closed.

Parameters **`callback_method`** (*method*) – The callback when the channel is opened

[1] "more effective flow control mechanism that does not require cooperation from clients and reacts quickly to prevent the broker from exhausing memory - see http://www.rabbitmq.com/extensions.html#memsup" from http://lists.rabbitmq.com/pipermail/rabbitmq-announce/attachments/20100825/2c672695/attachment.txt

add_on_open_callback(*callback_method*)
Add a callback notification when the connection has opened.

Parameters **callback_method** (*method*) – The callback when the channel is opened

add_timeout(*deadline*, *callback_method*)
Add the callback_method to the IOLoop timer to fire after deadline seconds. Returns a handle to the timeout

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback
- **callback_method** (*method*) – The callback method

Return type str

basic_nack
Specifies if the server supports basic.nack on the active connection.

Return type bool

channel(*on_open_callback*, *channel_number=None*)
Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters

- **on_open_callback** (*method*) – The callback when the channel is opened
- **channel_number** (*int*) – The channel number to use, defaults to the next available.

close(*reply_code=200*, *reply_text='Normal shutdown'*)
Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

consumer_cancel_notify
Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

exchange_exchange_bindings
Specifies if the active connection supports exchange to exchange bindings.

Return type bool

is_closed
Returns a boolean reporting the current connection state.

is_closing
Returns a boolean reporting the current connection state.

is_open
Returns a boolean reporting the current connection state.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

remove_timeout(*timeout_id*)
Remove the timeout from the IOLoop by the ID returned from add_timeout.

Return type str

set_backpressure_multiplier(*value=10*)
Alter the backpressure multiplier value. We set this to 10 by default. This value is used to raise warnings and trigger the backpressure callback.

Parameters **value** (*int*) – The multiplier value to set

BlockingConnection

The BlockingConnection creates a layer on top of Pika's asynchronous core providng methods that will block until their expected response has returned. Due to the asynchronous nature of the Basic.Deliver and Basic.Return calls from RabbitMQ to your application, you are still required to implement continuation-passing style asynchronous methods if you'd like to receive messages from RabbitMQ using basic_consume or if you want to be notified of a delivery failure when using basic_publish.

Basic.Get is a blocking call which will either return the Method Frame, Header Frame and Body of a message, or it will return a Basic.GetEmpty frame as the Method Frame.

For more information on using the BlockingConnection, see `BlockingChannel`

Publishing Example:

```
from pika.adapters import BlockingConnection
from pika import BasicProperties

# Open a connection to RabbitMQ on localhost using all default parameters
connection = BlockingConnection()

# Open the channel
channel = connection.channel()

# Declare the queue
channel.queue_declare(queue="test", durable=True, exclusive=False, auto_delete=False)

# Send a message
channel.basic_publish(exchange='',
                      routing_key="test",
                      body="Hello World!",
                      properties=BasicProperties(content_type="text/plain",
                                                 delivery_mode=1))
```

Consuming Example:

```
from pika.adapters import BlockingConnection

# Open a connection to RabbitMQ on localhost using all default parameters
connection = BlockingConnection()

# Open the channel
channel = connection.channel()

# Declare the queue
channel.queue_declare(queue="test", durable=True,
                      exclusive=False, auto_delete=False)
```

```
# Start our counter at 0
messages = 0

# Method that will receive our messages and stop consuming after 10
def _on_message(channel, method, header, body):
    print "Message:"
    print "\t%r" % method
    print "\t%r" % header
    print "\t%r" % body

    # Acknowledge message receipt
    channel.basic_ack(method.delivery_tag)

    # We've received 10 messages, stop consuming
    global messages
    messages += 1
    if messages > 10:
        channel.stop_consuming()

# Setup up our consumer callback
channel.basic_consume(_on_message, queue="test")

# This is blocking until channel.stop_consuming is called and will allow us to receive messages
channel.start_consuming()
```

Implement a blocking, procedural style connection adapter on top of the asynchronous core.

class pika.adapters.blocking_connection.**BlockingConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=True*)

The BlockingConnection adapter is meant for simple implementations where you want to have blocking behavior. The behavior layered on top of the async library. Because of the nature of AMQP there are a few callbacks one needs to do, even in a blocking implementation. These include receiving messages from Basic.Deliver, Basic.GetOk, and Basic.Return.

add_timeout(*deadline*, *callback*)

Add the callback to the IOLoop timer to fire after deadline seconds.

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback
- **callback** (*method*) – The callback method

Return type str

channel(*channel_number=None*)

Create a new channel with the next available or specified channel #.

Parameters **channel_number** (*int*) – Specify the channel number

close(*reply_code=200*, *reply_text='Normal shutdown'*)

Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

disconnect()
Disconnect from the socket

process_data_events()
Will make sure that data events are processed. Your app can block on this method.

process_timeouts()
Process the self._timeouts event stack

remove_timeout(*timeout_id*)
Remove the timeout from the IOLoop by the ID returned from add_timeout.

Parameters **timeout_id** (*str*) – The id of the timeout to remove

send_method(*channel_number*, *method_frame*, *content=None*)
Constructs a RPC method frame and then sends it to the broker.

Parameters

- **channel_number** (*int*) – The channel number for the frame
- **method_frame** (*pika.object.Method*) – The method frame to send
- **content** (*tuple*) – If set, is a content frame, is tuple of properties and body.

sleep(*duration*)
A safer way to sleep than calling time.sleep() directly which will keep the adapter from ignoring frames sent from RabbitMQ. The connection will "sleep" or block the number of seconds specified in duration in small intervals.

Parameters **duration** (*int*) – The time to sleep

add_backpressure_callback(*callback_method*)
Call method "callback" when pika believes backpressure is being applied.

Parameters **callback_method** (*method*) – The method to call

add_on_close_callback(*callback_method*)
Add a callback notification when the connection has closed.

Parameters **callback_method** (*method*) – The callback when the channel is opened

add_on_open_callback(*callback_method*)
Add a callback notification when the connection has opened.

Parameters **callback_method** (*method*) – The callback when the channel is opened

basic_nack
Specifies if the server supports basic.nack on the active connection.

Return type bool

consumer_cancel_notify
Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

exchange_exchange_bindings
Specifies if the active connection supports exchange to exchange bindings.

Return type bool

is_closed
Returns a boolean reporting the current connection state.

is_closing
Returns a boolean reporting the current connection state.

is_open
Returns a boolean reporting the current connection state.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

set_backpressure_multiplier(*value=10*)
Alter the backpressure multiplier value. We set this to 10 by default. This value is used to raise warnings and trigger the backpressure callback.

Parameters **value** (*int*) – The multiplier value to set

SelectConnection

Note: SelectConnection is the recommended method for using Pika under most circumstances. It supports multiple event notification methods including select, epoll, kqueue and poll.

By default SelectConnection will attempt to use the most appropriate event notification method for your system. In order to override the default behavior you may set the poller type by assigning a string value to the select_connection modules POLLER_TYPE attribute prior to creating the SelectConnection object instance. Valid values are: kqueue, poll, epoll, select

Poller Type Override Example:

```
import select_connection
select_connection.POLLER_TYPE = 'epoll'
connection = select_connection.SelectConnection()
```

See the *Continuation-Passing Style example* for an example of using SelectConnection. A connection adapter that tries to use the best polling method for the platform pika is running on.

class pika.adapters.select_connection.**SelectConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=True*)
An asynchronous connection adapter that attempts to use the fastest event loop adapter for the given platform.

add_backpressure_callback(*callback_method*)
Call method "callback" when pika believes backpressure is being applied.

Parameters **callback_method** (*method*) – The method to call

add_on_close_callback(*callback_method*)
Add a callback notification when the connection has closed.

Parameters **callback_method** (*method*) – The callback when the channel is opened

add_on_open_callback(*callback_method*)
Add a callback notification when the connection has opened.

Parameters **callback_method** (*method*) – The callback when the channel is opened

add_timeout(*deadline*, *callback_method*)
Add the callback_method to the IOLoop timer to fire after deadline seconds. Returns a handle to the timeout

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback
- **callback_method** (*method*) – The callback method

Return type str

basic_nack
Specifies if the server supports basic.nack on the active connection.

Return type bool

channel(*on_open_callback*, *channel_number=None*)
Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters

- **on_open_callback** (*method*) – The callback when the channel is opened
- **channel_number** (*int*) – The channel number to use, defaults to the next available.

close(*reply_code=200*, *reply_text='Normal shutdown'*)
Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

consumer_cancel_notify
Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

exchange_exchange_bindings
Specifies if the active connection supports exchange to exchange bindings.

Return type bool

is_closed
Returns a boolean reporting the current connection state.

is_closing
Returns a boolean reporting the current connection state.

is_open
Returns a boolean reporting the current connection state.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

remove_timeout(*timeout_id*)
Remove the timeout from the IOLoop by the ID returned from add_timeout.

Return type str

set_backpressure_multiplier(*value=10*)
Alter the backpressure multiplier value. We set this to 10 by default. This value is used to raise warnings and trigger the backpressure callback.

Parameters **value** (*int*) – The multiplier value to set

TornadoConnection

Tornado is an open source version of the scalable, non-blocking web server and tools that power FriendFeed. For more information on tornado, visit http://tornadoweb.org

Since the Tornado IOLoop blocks once it is started, it is suggested that you use a timer to add Pika to your tornado.Application instance after the HTTPServer has started.

The following is a simple, non-working example on how to add Pika to the Tornado IOLoop without blocking other applications from doing so. To see a fully workng example, see the Tornado Demo application in the examples.

Example:

```
from pika.adapters.tornado_connection import TornadoConnection

class PikaClient(object):
    def connect(self):
        self.connection = TornadoConnection(on_connected_callback=self.on_connected)

# Create our Tornado Application
application = tornado.web.Application([
    (r"/", ExampleHandler)
], **settings)

# Create our Pika Client
application.pika = PikaClient()

# Start the HTTPServer
http_server = tornado.httpserver.HTTPServer(application)
http_server.listen(8080)

# Get a handle to the instance of IOLoop
ioloop = tornado.ioloop.IOLoop.instance()

# Add our Pika connect to the IOLoop since we loop on ioloop.start
ioloop.add_timeout(500, application.pika.connect)

# Start the IOLoop
ioloop.start()
```

Run pika on the Tornado IOLoop

class pika.adapters.tornado_connection.**TornadoConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=False*, *custom_ioloop=None*)

The TornadoConnection runs on the Tornado IOLoop. If you're running the connection in a web app, make sure you set stop_ioloop_on_close to False, which is the default behavior for this adapter, otherwise the web app will stop taking requests.

add_timeout(*deadline*, *callback_method*)

Add the callback_method to the IOLoop timer to fire after deadline seconds. Returns a handle to the timeout. Do not confuse with Tornado's timeout where you pass in the time you want to have your callback called. Only pass in the seconds until it's to be called.

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback

- **callback_method** (*method*) – The callback method

Return type str

remove_timeout(*timeout_id*)
Remove the timeout from the IOLoop by the ID returned from add_timeout.

Return type str

add_backpressure_callback(*callback_method*)
Call method "callback" when pika believes backpressure is being applied.

Parameters **callback_method** (*method*) – The method to call

add_on_close_callback(*callback_method*)
Add a callback notification when the connection has closed.

Parameters **callback_method** (*method*) – The callback when the channel is opened

add_on_open_callback(*callback_method*)
Add a callback notification when the connection has opened.

Parameters **callback_method** (*method*) – The callback when the channel is opened

basic_nack
Specifies if the server supports basic.nack on the active connection.

Return type bool

channel(*on_open_callback*, *channel_number=None*)
Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters

- **on_open_callback** (*method*) – The callback when the channel is opened
- **channel_number** (*int*) – The channel number to use, defaults to the next available.

close(*reply_code=200*, *reply_text='Normal shutdown'*)
Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

consumer_cancel_notify
Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

exchange_exchange_bindings
Specifies if the active connection supports exchange to exchange bindings.

Return type bool

is_closed
Returns a boolean reporting the current connection state.

is_closing
Returns a boolean reporting the current connection state.

is_open
Returns a boolean reporting the current connection state.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

set_backpressure_multiplier(*value=10*)
Alter the backpressure multiplier value. We set this to 10 by default. This value is used to raise warnings and trigger the backpressure callback.

Parameters **value** (*int*) – The multiplier value to set

2.2 Usage Examples

Pika has various methods of use, between the synchronous BlockingConnection adapter and the various asynchronous connection adapter. The following examples illustrate the various ways that you can use Pika in your projects.

2.2.1 Using URLParameters

Pika has two methods of encapsulating the data that lets it know how to connect to RabbitMQ, *pika.connection.ConnectionParameters()* and *pika.connection.URLParameters()*.

Note: If you're connecting to RabbitMQ on localhost on port 5672, with the default virtual host of / and the default username and password of *guest* and *guest*, you do not need to specify connection parameters when connecting.

Using *pika.connection.URLParameters()* is an easy way to minimize the variables required to connect to RabbitMQ and supports all of the directives that *pika.connection.ConnectionParameters()* supports.

The following is the format for the URLParameters connection value:

```
scheme://username:password@host:port/virtual_host?key=value&key=value
```

As you can see, by default, the scheme (amqp, amqps), username, password, host, port and virtual host make up the core of the URL and any other parameter is passed in as query string values.

Example Connection URLS

The default connection URL connects to the / virtual host as guest using the guest password on localhost port 5672. Note the forwardslash in the URL is encoded to %2F:

```
amqp://guest:guest@localhost:5672/%2F
```

Connect to a host *rabbit1* as the user *www-data* using the password *rabbit_pwd* on the virtual host *web_messages*:

```
amqp://www-data:rabbit_pwd@rabbit1/web_messages
```

Connecting via SSL is pretty easy too. To connect via SSL for the previous example, simply change the scheme to *amqps*. If you do not specify a port, Pika will use the default SSL port of 5671:

```
amqp://www-data:rabbit_pwd@rabbit1/web_messages
```

If you're looking to tweak other parameters, such as enabling heartbeats, simply add the key/value pair as a query string value. The following builds upon the SSL connection, enabling heartbeats every 30 seconds:

```
amqp://www-data:rabbit_pwd@rabbit1/web_messages?heartbeat_interval=30
```

Options that are available as query string values:

- backpressure_detection: Pass in a value of *t* to enable backpressure detection, it is disabled by default.
- channel_max: Alter the default channel maximum by passing in a 32-bit integer value here
- connection_attempts: Alter the default of 1 connection attempt by passing in an integer value here [1].
- frame_max: Alter the default frame maximum size value by passing in a long integer value [2].
- heartbeat_interval: Pass a value greater than zero to enable heartbeats between the server and your application. The integer value you pass here will be the number of seconds between heartbeats.
- locale: Set the locale of the client using underscore delimited posix Locale code in ll_CC format (en_US, pt_BR, de_DE).
- retry_delay: The number of seconds to wait before attempting to reconnect on a failed connection, if connection_attempts is > 0.
- socket_timeout: Change the default socket timeout duration from 0.25 seconds to another integer or float value. Adjust with caution.
- **ssl_options: A url encoded dict of values for the SSL connection. The available keys are:**
  - ca_certs
  - cert_reqs
  - certfile
  - keyfile
  - ssl_version

For an information on what the ssl_options can be set to reference the official Python documentation. Here is an example of setting the client certificate and key:

```
amqp://www-data:rabbit_pwd@rabbit1/web_messages?heartbeat_interval=30&ssl_options=%7B%27keyfile%27%3A
```

The following example demonstrates how to generate the ssl_options string with Python's urllib:

```
import urllib
urllib.urlencode({'ssl_options': {'certfile': '/etc/ssl/mycert.pem', 'keyfile': '/etc/ssl/mykey.pem'}
```

2.2.2 Connecting to RabbitMQ with Callback-Passing Style

When you connect to RabbitMQ with an asynchronous adapter, you are writing event oriented code. The connection adapter will block on the IOLoop that is watching to see when pika should read data from and write data to RabbitMQ. Because you're now blocking on the IOLoop, you will receive callback notifications when specific events happen.

[1] The *pika.adapters.blocking_connection.BlockingConnection()* adapter does not respect the *connection_attempts* parameter.

[2] The AMQP specification states that a server can reject a request for a frame size larger than the value it passes during content negotiation.

Example Code

In the example, there are three steps that take place:

1. Setup the connection to RabbitMQ
2. Start the IOLoop
3. Once connected, the on_open method will be called by Pika with a handle to the connection. In this method, a new channel will be opened on the connection.
4. Once the channel is opened, you can do your other actions, whether they be publishing messages, consuming messages or other RabbitMQ related activities.:

```
import pika

# Step #3
def on_open(connection):
    connection.channel(on_channel_open)

# Step #4
def on_channel_open(channel):
    channel.basic_publish('exchange_name',
                          'routing_key',
                          'Test Message',
                          pika.BasicProperties(content_type='text/plain',
                                               type='example'))

# Step #1: Connect to RabbitMQ
connection = pika.SelectConnection(on_open_callback=on_open)

try:
    # Step #2 - Block on the IOLoop
    connection.ioloop.start()

# Catch a Keyboard Interrupt to make sure that the connection is closed cleanly
except KeyboardInterrupt:

    # Gracefully close the connection
    connection.close()

    # Start the IOLoop again so Pika can communicate, it will stop on its own when the connectio
    connection.ioloop.start()
```

2.2.3 Using the Blocking Connection to get a message from RabbitMQ

The `BlockingChannel.basic_get()` method will return a tuple with the members.

If the server return a message, the first item in the tuple will be a *pika.spec.Basic.GetOk()* object with the current message count, the redelivered flag, the routing key that was used to put the message in the queue, and the exchange the message was published to. The second item will be a `pika.spec.Basic.Properties()` object and the third will be the message body.

If the server did not return a message a tuple of None, None, None will be returned.

Example of getting a message and acknowledging it:

```
import pika
```

```
connection = pika.BlockingConnection()
channel = connection.channel()
method_frame, header_frame, body = channel.basic_get('test')
if method_frame:
    print method_frame, header_frame, body
    channel.basic_ack(method_frame.delivery_tag)
else:
    print 'No message returned'
```

2.2.4 Comparing Message Publishing with BlockingConnection and SelectConnection

For those doing simple, non-asynchronous programing, *pika.adapters.blocking_connection.BlockingConnection()* proves to be the easiest way to get up and running with Pika to publish messages.

In the following example, a connection is made to RabbitMQ listening to port *5672* on *localhost* using the username *guest* and password *guest* and virtual host */*. Once connected, a channel is opened and a message is published to the *test_exchange* exchange using the *test_routing_key* routing key. The BasicProperties value passed in sets the message to delivery mode *1* (non-persisted) with a content-type of *text/plain*. Once the message is published, the connection is closed:

```
import pika

parameters = pika.URLParameters('amqp://guest:guest@localhost:5672/%2F')

connection = pika.BlockingConnection(paramters)

channel = connection.channel()

channel.basic_publish('test_exchange',
                      'test_routing_key',
                      'message body value',
                      pika.BasicProperties(content_type='text/plain',
                                           delivery_mode=1))

connection.close()
```

In contrast, using *pika.adapters.select_connection.SelectConnection()* and the other asynchronous adapters is more complicated and less pythonic, but when used with other asynchronous services can have tremendous performance improvements. In the following code example, all of the same parameters and values are used as were used in the previous example:

```
import pika

# Step #3
def on_open(connection):

    connection.channel(on_channel_open)

# Step #4
def on_channel_open(channel):

    channel.basic_publish('test_exchange',
                          'test_routing_key',
                          'message body value',
                          pika.BasicProperties(content_type='text/plain',
```

```
                                                delivery_mode=1))

    connection.close()

# Step #1: Connect to RabbitMQ
parameters = pika.URLParameters('amqp://guest:guest@localhost:5672/%2F')

connection = pika.SelectConnection(parameters=parameters,
                                   on_open_callback=on_open)

try:

    # Step #2 - Block on the IOLoop
    connection.ioloop.start()

# Catch a Keyboard Interrupt to make sure that the connection is closed cleanly
except KeyboardInterrupt:

    # Gracefully close the connection
    connection.close()

    # Start the IOLoop again so Pika can communicate, it will stop on its own when the connection is
    connection.ioloop.start()
```

2.3 Frequently Asked Questions

- Is Pika thread safe?

  Pika does not have any notion of threading in the code. If you want to use Pika with threading, make sure you have a Pika connection per thread, created in that thread. It is not safe to share one Pika connection across threads.

- How do I report a bug with Pika?

  The main Pika repository is hosted on Github and we use the Issue tracker at https://github.com/pika/pika/issues.

- Is there a mailing list for Pika?

  Yes, Pika's mailing list is available on Google Groups and the email address is pika-python@googlegroups.com, though traditionally questions about Pika have been asked on the RabbitMQ-Discuss mailing list.

- Is there an IRC channel for Pika?

  People knowledgeable about Pika tend to hang out in #pika and #RabbitMQ on irc.freenode.net.

- What versions of Python are supported?

  Main developent is currently on the Python 2.6 branch. For a release, Pika passes its tests on the latest versions of 2.5, 2.6 and 2.7.

- Does Python work with Python 3?

  Not yet.

- How can I contribute to Pika?

  You can fork the project on Github and issue Pull Requests when you believe you have something solid to be added to the main repository.

CHAPTER 3

0.9.6 Release Notes

- New features
  - URLParameters
  - BlockingChannel.start_consuming() and BlockingChannel.stop_consuming()
  - Delivery Confirmations
  - Improved unittests
- Major bugfix areas
  - Connection handling
  - Blocking functionality in the BlockingConnection
  - SSL
  - UTF-8 Handling
- Removals
  - pika.reconnection_strategies
  - pika.channel.ChannelTransport
  - pika.log
  - pika.template
  - examples directory
- Over 700 commits from 37 contributors

CHAPTER 4

Pika Core Modules and Classes

Note: The following documentation is for Pika development and is not intended to be end-user documentation.

4.1 adapters

Note: The following class level documentation is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself. For an overview of how to use adapters, please reference the Connecting to RabbitMQ documentation.

4.1.1 base_connection

Base class extended by connection adapters. This extends the connection.Connection class to encapsulate connection behavior but still isolate socket and low level communication.

BaseConnection

class pika.adapters.base_connection.**BaseConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=True*)

BaseConnection class that should be extended by connection adapters

READ = 1

WRITE = 4

ERROR = 8

ERRORS_TO_IGNORE = [11, 11, 4]

DO_HANDSHAKE = True

WARN_ABOUT_IOLOOP = False

add_timeout(*deadline*, *callback_method*)

Add the callback_method to the IOLoop timer to fire after deadline seconds. Returns a handle to the timeout

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback
- **callback_method** (*method*) – The callback method

Return type str

close(*reply_code=200*, *reply_text='Normal shutdown'*)
Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

remove_timeout(*timeout_id*)
Remove the timeout from the IOLoop by the ID returned from add_timeout.

Return type str

_adapter_connect()
Connect to the RabbitMQ broker

_adapter_disconnect()
Invoked if the connection is being told to disconnect

_check_state_on_disconnect()
Checks to see if we were in opening a connection with RabbitMQ when we were disconnected and raises exceptions for the anticipated exception types.

_create_and_connect_to_socket()
Create socket and connect to it, using SSL if enabled.

_do_ssl_handshake()
Perform SSL handshaking, copied from python stdlib test_ssl.py.

_get_error_code(*error_value*)
Get the error code from the error_value accounting for Python version differences.

Return type int

_flush_outbound()
Call the state manager who will figure out that we need to write.

_handle_disconnect()
Called internally when the socket is disconnected already

_handle_ioloop_stop()
Invoked when the connection is closed to determine if the IOLoop should be stopped or not.

_handle_error(*error_value*)
Internal error handling method. Here we expect a socket.error coming in and will handle different socket errors differently.

Parameters **error_value** (*int|object*) – The inbound error

_handle_events(*fd*, *events*, *error=None*, *write_only=False*)
Handle IO/Event loop events, processing them.

Parameters

- **fd** (*int*) – The file descriptor for the events

- **events** (*int*) – Events from the IO/Event loop
- **error** (*int*) – Was an error specified
- **write_only** (*bool*) – Only handle write events

_handle_read()
Read from the socket and call our on_data_available with the data.

_handle_write()
Handle any outbound buffer writes that need to take place.

_init_connection_state()
Initialize or reset all of our internal state variables for a given connection. If we disconnect and reconnect, all of our state needs to be wiped.

_manage_event_state()
Manage the bitmask for reading/writing/error which is used by the io/event handler to specify when there is an event such as a read or write.

_wrap_socket(*sock*)
Wrap the socket for connecting over SSL.

Return type ssl.SSLSocket

4.1.2 asyncore_connection

Use pika with the stdlib asyncore module

AsyncoreConnection

class pika.adapters.asyncore_connection.**AsyncoreConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=True*)

_adapter_connect()
Connect to our RabbitMQ broker using AsyncoreDispatcher, then setting Pika's suggested buffer size for socket reading and writing. We pass the handle to self so that the AsyncoreDispatcher object can call back into our various state methods.

PikaDispatcher

class pika.adapters.asyncore_connection.**PikaDispatcher**(*sock=None*, *map=None*, *event_callback=None*)

READ = 1

WRITE = 4

ERROR = 8

add_timeout(*deadline*, *handler*)
Add a timeout with with given deadline, should return a timeout id.

Parameters

- **deadline** (*int*) – The number of seconds to wait until calling handler
- **handler** (*method*) – The method to call at deadline

Return type str

readable()

writable()

handle_read()

handle_write()

process_timeouts()
Process the self._timeouts event stack

remove_timeout(*timeout_id*)
Remove a timeout if it's still in the timeout stack

Parameters **timeout_id** (*str*) – The timeout id to remove

start()

stop()

update_handler(*fileno_unused*, *events*)
Set the events to the current events

Parameters

- **fileno_unused** (*int*) – The file descriptor
- **events** (*int*) – The event mask

4.1.3 blocking_connection

Implement a blocking, procedural style connection adapter on top of the asynchronous core.

BlockingConnection

class pika.adapters.blocking_connection.**BlockingConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=True*)
The BlockingConnection adapter is meant for simple implementations where you want to have blocking behavior. The behavior layered on top of the async library. Because of the nature of AMQP there are a few callbacks one needs to do, even in a blocking implementation. These include receiving messages from Basic.Deliver, Basic.GetOk, and Basic.Return.

WRITE_TO_READ_RATIO = 1000

DO_HANDSHAKE = True

SLEEP_DURATION = 0.1

SOCKET_CONNECT_TIMEOUT = 0.25

SOCKET_TIMEOUT_THRESHOLD = 12

SOCKET_TIMEOUT_CLOSE_THRESHOLD = 3

SOCKET_TIMEOUT_MESSAGE = 'Timeout exceeded, disconnected'

add_timeout(*deadline*, *callback*)
Add the callback to the IOLoop timer to fire after deadline seconds.

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback
- **callback** (*method*) – The callback method

Return type str

channel(*channel_number=None*)
Create a new channel with the next available or specified channel #.

Parameters **channel_number** (*int*) – Specify the channel number

close(*reply_code=200*, *reply_text='Normal shutdown'*)
Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

disconnect()
Disconnect from the socket

process_data_events()
Will make sure that data events are processed. Your app can block on this method.

process_timeouts()
Process the self._timeouts event stack

remove_timeout(*timeout_id*)
Remove the timeout from the IOLoop by the ID returned from add_timeout.

Parameters **timeout_id** (*str*) – The id of the timeout to remove

send_method(*channel_number*, *method_frame*, *content=None*)
Constructs a RPC method frame and then sends it to the broker.

Parameters

- **channel_number** (*int*) – The channel number for the frame
- **method_frame** (*pika.object.Method*) – The method frame to send
- **content** (*tuple*) – If set, is a content frame, is tuple of properties and body.

sleep(*duration*)
A safer way to sleep than calling time.sleep() directly which will keep the adapter from ignoring frames sent from RabbitMQ. The connection will "sleep" or block the number of seconds specified in duration in small intervals.

Parameters **duration** (*int*) – The time to sleep

_adapter_connect()
Connect to the RabbitMQ broker

_adapter_disconnect()
Called if the connection is being requested to disconnect.

_call_timeout_method(*timeout_value*)
Execute the method that was scheduled to be called.

Parameters **timeout_value** (*dict*) – The configuration for the timeout

_deadline_passed(*timeout_id*)
Returns True if the deadline has passed for the specified timeout_id.

Parameters **timeout_id** (*str*) – The id of the timeout to check

Return type bool

_handle_disconnect()
Called internally when the socket is disconnected already

_handle_read()

_handle_timeout()
Invoked whenever the socket times out

_flush_outbound()
Flush the outbound socket buffer.

_on_connection_closed(*method_frame*, *from_adapter=False*)
Called when the connection is closed remotely. The from_adapter value will be true if the connection adapter has been disconnected from the broker and the method was invoked directly instead of by receiving a Connection.Close frame.

Parameters

- **pika.frame.Method** – The Connection.Close frame
- **from_adapter** (*bool*) – Called by the connection adapter

Raises AMQPConnectionError

_send_frame(*frame_value*)
This appends the fully generated frame to send to the broker to the output buffer which will be then sent via the connection adapter.

Parameters **frame_value** (*pika.frame.Frame|pika.frame.ProtocolHeader*) – The frame to write

4.1.4 select_connection

A connection adapter that tries to use the best polling method for the platform pika is running on.

SelectConnection

class pika.adapters.select_connection.**SelectConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=True*)
An asynchronous connection adapter that attempts to use the fastest event loop adapter for the given platform.

_adapter_connect()
Connect to the RabbitMQ broker

_flush_outbound()
Call the state manager who will figure out that we need to write then call the poller's poll function to force it to process events.

IOLoop

class pika.adapters.select_connection.**IOLoop**(*state_manager*)

Singlton wrapper that decides which type of poller to use, creates an instance of it in start_poller and keeps the invoking application in a blocking state by calling the pollers start method. Poller should keep looping until IOLoop.instance().stop() is called or there is a socket error.

Also provides a convenient pass-through for add_timeout and set_events

add_timeout(*deadline*, *handler*)

Add a timeout with with given deadline, should return a timeout id.

Pass through a deadline and handler to the active poller.

Parameters

- **deadline** (*int*) – The number of seconds to wait until calling handler
- **handler** (*method*) – The method to call at deadline

Return type int

poller_type

Return the type of poller.

Return type str

remove_timeout(*timeout_id*)

Remove a timeout if it's still in the timeout stack of the poller

Parameters **timeout_id** (*str*) – The timeout id to remove

start()

Start the IOLoop, waiting for a Poller to take over.

start_poller(*handler*, *events*, *fileno*)

Start the Poller, once started will take over for IOLoop.start()

Parameters

- **handler** (*method*) – The method to call to handle events
- **events** (*int*) – The events to handle
- **fileno** (*int*) – The file descriptor to poll for

stop()

Stop the poller's event loop

update_handler(*fileno*, *events*)

Pass in the events to process for the given file descriptor.

Parameters

- **fileno** (*int*) – The file descriptor to poll for
- **events** (*int*) – The events to handle

SelectPoller

class pika.adapters.select_connection.**SelectPoller**(*fileno*, *handler*, *events*, *state_manager*)

Default behavior is to use Select since it's the widest supported and has all of the methods we need for child classes as well. One should only need to override the update_handler and start methods for additional types.

TIMEOUT = 1

add_timeout(*deadline*, *handler*)
Add a timeout with with given deadline, should return a timeout id.

Parameters

- **deadline** (*int*) – The number of seconds to wait until calling handler
- **handler** (*method*) – The method to call at deadline

Return type str

flush_pending_timeouts()

poll(*write_only=False*)
Check to see if the events that are cared about have fired.

Parameters **write_only** (*bool*) – Don't look at self.events, just look to see if the adapter can write.

process_timeouts()
Process the self._timeouts event stack

remove_timeout(*timeout_id*)
Remove a timeout if it's still in the timeout stack

Parameters **timeout_id** (*str*) – The timeout id to remove

start()
Start the main poller loop. It will loop here until self.closed

update_handler(*fileno*, *events*)
Set the events to the current events

Parameters

- **fileno** (*int*) – The file descriptor
- **events** (*int*) – The event mask

KQueuePoller

class pika.adapters.select_connection.**KQueuePoller**(*fileno*, *handler*, *events*, *state_manager*)
KQueuePoller works on BSD based systems and is faster than select

update_handler(*fileno*, *events*)
Set the events to the current events

Parameters

- **fileno** (*int*) – The file descriptor
- **events** (*int*) – The event mask

start()
Start the main poller loop. It will loop here until self.closed

poll(*write_only=False*)
Check to see if the events that are cared about have fired.

Parameters **write_only** (*bool*) – Don't look at self.events, just look to see if the adapter can write.

TIMEOUT = 1

add_timeout(*deadline*, *handler*)
Add a timeout with with given deadline, should return a timeout id.

Parameters

- **deadline** (*int*) – The number of seconds to wait until calling handler
- **handler** (*method*) – The method to call at deadline

Return type str

flush_pending_timeouts()

process_timeouts()
Process the self._timeouts event stack

remove_timeout(*timeout_id*)
Remove a timeout if it's still in the timeout stack

Parameters **timeout_id** (*str*) – The timeout id to remove

PollPoller

class pika.adapters.select_connection.**PollPoller**(*fileno*, *handler*, *events*, *state_manager*)
Poll works on Linux and can have better performance than EPoll in certain scenarios. Both are faster than select.

update_handler(*fileno*, *events*)
Set the events to the current events

Parameters

- **fileno** (*int*) – The file descriptor
- **events** (*int*) – The event mask

start()
Start the main poller loop. It will loop here until self.closed

poll(*write_only=False*)
Poll until TIMEOUT waiting for an event

Parameters **write_only** (*bool*) – Only process write events

TIMEOUT = 1

add_timeout(*deadline*, *handler*)
Add a timeout with with given deadline, should return a timeout id.

Parameters

- **deadline** (*int*) – The number of seconds to wait until calling handler
- **handler** (*method*) – The method to call at deadline

Return type str

flush_pending_timeouts()

process_timeouts()
Process the self._timeouts event stack

remove_timeout(*timeout_id*)
Remove a timeout if it's still in the timeout stack

Parameters **timeout_id** (*str*) – The timeout id to remove

EPollPoller

class pika.adapters.select_connection.**EPollPoller**(*fileno*, *handler*, *events*, *state_manager*)
EPoll works on Linux and can have better performance than Poll in certain scenarios. Both are faster than select.

poll(*write_only=False*)
Poll until TIMEOUT waiting for an event

Parameters **write_only** (*bool*) – Only process write events

TIMEOUT = 1

add_timeout(*deadline*, *handler*)
Add a timeout with with given deadline, should return a timeout id.

Parameters

- **deadline** (*int*) – The number of seconds to wait until calling handler
- **handler** (*method*) – The method to call at deadline

Return type str

flush_pending_timeouts()

process_timeouts()
Process the self._timeouts event stack

remove_timeout(*timeout_id*)
Remove a timeout if it's still in the timeout stack

Parameters **timeout_id** (*str*) – The timeout id to remove

start()
Start the main poller loop. It will loop here until self.closed

update_handler(*fileno*, *events*)
Set the events to the current events

Parameters

- **fileno** (*int*) – The file descriptor
- **events** (*int*) – The event mask

4.1.5 tornado_connection

Run pika on the Tornado IOLoop

TornadoConnection

class pika.adapters.tornado_connection.**TornadoConnection**(*parameters=None*, *on_open_callback=None*, *stop_ioloop_on_close=False*, *custom_ioloop=None*)
The TornadoConnection runs on the Tornado IOLoop. If you're running the connection in a web app, make sure

you set stop_ioloop_on_close to False, which is the default behavior for this adapter, otherwise the web app will stop taking requests.

WARN_ABOUT_IOLOOP = True

_adapter_connect()
Connect to the RabbitMQ broker

_adapter_disconnect()
Disconnect from the RabbitMQ broker

add_timeout(*deadline*, *callback_method*)
Add the callback_method to the IOLoop timer to fire after deadline seconds. Returns a handle to the timeout. Do not confuse with Tornado's timeout where you pass in the time you want to have your callback called. Only pass in the seconds until it's to be called.

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback
- **callback_method** (*method*) – The callback method

Return type str

remove_timeout(*timeout_id*)
Remove the timeout from the IOLoop by the ID returned from add_timeout.

Return type str

4.2 amqp_object

Note: This class is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

The class in this module, amqp_object.AMQPObject extends Python's object class creating a base class that other classes, who would like an easy to implement representation of the class state, may extend. Base classes that are extended by low level AMQP frames and higher level AMQP classes and methods.

4.2.1 object

class pika.amqp_object.**AMQPObject**
Base object that is extended by AMQP low level frames and AMQP classes and methods.

NAME = 'AMQPObject'

INDEX = None

4.2.2 Class

class pika.amqp_object.**Class**
Is extended by AMQP classes

NAME = 'Unextended Class'

INDEX = None

4.2.3 Method

class pika.amqp_object.**Method**
Is extended by AMQP methods

NAME = 'Unextended Method'

synchronous = False

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.

Parameters

- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_properties()
Return the properties if they are set.

Return type pika.frame.Properties

get_body()
Return the message body if it is set.

Return type str|unicode

INDEX = None

4.2.4 Properties

class pika.amqp_object.**Properties**
Class to encompass message properties (AMQP Basic.Properties)

NAME = 'Unextended Properties'

INDEX = None

4.3 callback

Note: The following class is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

Callback management class, common area for keeping track of all callbacks in the Pika stack.

4.3.1 CallbackManager

class pika.callback.**CallbackManager**
CallbackManager is a global callback system designed to be a single place where Pika can manage callbacks and process them. It should be referenced by the CallbackManager.instance() method instead of constructing new instances of it.

add(**args*, ***kwargs*)
Add a callback to the stack for the specified key. If the call is specified as one_shot, it will be removed after being fired

The prefix is usually the channel number but the class is generic and prefix and key may be any value. If you pass in only_caller CallbackManager will restrict processing of the callback to only the calling function/object that you specify.

Parameters

- **prefix** (*str|int*) – Categorize the callback
- **key** (*object|str|dict*) – The key for the callback
- **callback** (*method*) – The callback to call
- **one_shot** (*bool*) – Remove this callback after it is called
- **only_caller** (*object*) – Only allow one_caller value to call the event that fires the callback.

Return type tuple(prefix, key)

cleanup(**args*, ***kwargs*)

Remove all callbacks from the stack by a prefix. Returns True if keys were there to be removed

Parameters **prefix** (*str*) – The prefix for keeping track of callbacks with

Return type bool

clear()

Clear all the callbacks if there are any defined.

pending(**args*, ***kwargs*)

Return count of callbacks for a given prefix or key or None

Parameters

- **prefix** (*str|int*) – Categorize the callback
- **key** (*Object|str|dict*) – The key for the callback

Return type None or int

process(**args*, ***kwargs*)

Run through and process all the callbacks for the specified keys. Caller should be specified at all times so that callbacks which require a specific function to call CallbackManager.process will not be processed.

Parameters

- **prefix** (*str|int*) – Categorize the callback
- **key** (*Object|str|dict*) – The key for the callback
- **caller** (*Object*) – Who is firing the event
- **args** (*list*) – Any optional arguments
- **keywords** (*dict*) – Optional keyword arguments

Return type bool

remove(**args*, ***kwargs*)

Remove a callback from the stack by prefix, key and optionally the callback itself. If you only pass in prefix and key, all callbacks for that prefix and key will be removed.

Parameters

- **prefix** (*str*) – The prefix for keeping track of callbacks with
- **key** (*str*) – The callback key

- **callback_value** (*method*) – The method defined to call on callback

Return type bool

4.4 channel

The Channel class provides a wrapper for interacting with RabbitMQ implementing the methods and behaviors for an AMQP Channel.

4.4.1 Channel

class pika.channel.**Channel**(*connection*, *channel_number*, *on_open_callback=None*)

A Channel is the primary communication method for interacting with RabbitMQ. It is recommended that you do not directly invoke the creation of a channel object in your application code but rather construct the a channel by calling the active connection's channel() method.

CLOSED = 0

OPENING = 1

OPEN = 2

CLOSING = 3

add_callback(*callback*, *replies*, *one_shot=True*)

Pass in a callback handler and a list replies from the RabbitMQ broker which you'd like the callback notified of. Callbacks should allow for the frame parameter to be passed in.

Parameters

- **callback** (*method*) – The method to call
- **replies** (*list*) – The replies to get a callback for
- **one_shot** (*bool*) – Only handle the first type callback

add_on_cancel_callback(*callback*)

Pass a callback function that will be called when the basic_cancel is sent by the server. The callback function should receive a frame parameter.

Parameters **callback** (*method*) – The method to call on callback

add_on_close_callback(*callback*)

Pass a callback function that will be called when the channel is closed. The callback function should receive a frame parameter.

Parameters **callback** (*method*) – The method to call on callback

add_on_flow_callback(*callback*)

Pass a callback function that will be called when Channel.Flow is called by the remote server. Note that newer versions of RabbitMQ will not issue this but instead use TCP backpressure

Parameters **callback** (*method*) – The method to call on callback

add_on_return_callback(*callback*)

Pass a callback function that will be called when basic_publish as sent a message that has been rejected and returned by the server. The callback handler should receive a method, header and body frame. The base signature for the callback should be the same as the method signature one creates for a basic_consume callback.

Parameters **callback** (*method*) – The method to call on callback

basic_ack(*delivery_tag=0*, *multiple=False*)
Acknowledge one or more messages. When sent by the client, this method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. When sent by server, this method acknowledges one or more messages published with the Publish method on a channel in confirm mode. The acknowledgement can be for a single message or a set of messages up to and including a specific message.

Parameters

- **delivery-tag** (*int*) – The server-assigned delivery tag
- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.

basic_cancel(*callback=None*, *consumer_tag=''*, *nowait=False*)
This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel-ok reply. It may also be sent from the server to the client in the event of the consumer being unexpectedly cancelled (i.e. cancelled for any reason other than the server receiving the corresponding basic.cancel from the client). This allows clients to be notified of the loss of consumers due to events such as queue deletion.

Parameters

- **callback** (*method*) – Method to call for a Basic.CancelOk response
- **consumer_tag** (*str*) – Identifier for the consumer
- **nowait** (*bool*) – Do not expect a Basic.CancelOk response

Raises ValueError

basic_consume(*consumer_callback*, *queue=''*, *no_ack=False*, *exclusive=False*, *consumer_tag=None*)
Sends the AMQP command Basic.Consume to the broker and binds messages for the consumer_tag to the consumer callback. If you do not pass in a consumer_tag, one will be automatically generated for you. Returns the consumer tag.

For more information on basic_consume, see: http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.consume

Parameters

- **consumer_callback** (*method*) – The method to callback when consuming
- **queue** (*str|unicode*) – The queue to consume from
- **no_ack** (*bool*) – Tell the broker to not expect a response
- **exclusive** (*bool*) – Don't allow other consumers on the queue
- **consumer_tag** (*str|unicode*) – Specify your own consumer tag

Return type str

basic_get(*callback=None*, *queue=''*, *no_ack=False*)
Get a single message from the AMQP broker. The callback method signature should have 3 parameters: The method frame, header frame and the body, like the consumer callback for Basic.Consume. If you want to be notified of Basic.GetEmpty, use the Channel.add_callback method adding your Basic.GetEmpty callback which should expect only one parameter, frame. For more information on basic_get and its parameters, see:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.get

Parameters

- **callback** (*method*) – The method to callback with a message
- **queue** (*str|unicode*) – The queue to get a message from
- **no_ack** (*bool*) – Tell the broker to not expect a reply

basic_nack(*delivery_tag=None*, *multiple=False*, *requeue=True*)

This method allows a client to reject one or more incoming messages. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery-tag** (*int*) – The server-assigned delivery tag
- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.
- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

basic_publish(*exchange*, *routing_key*, *body*, *properties=None*, *mandatory=False*, *immediate=False*)

Publish to the channel with the given exchange, routing key and body. For more information on basic_publish and what the parameters do, see:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.publish

Parameters

- **exchange** (*str*) – The exchange name
- **routing_key** (*str*) – The routing key
- **body** (*str*) – The message body
- **properties** (*pika.spec.Properties*) – Basic.properties
- **mandatory** (*bool*) – The mandatory flag
- **immediate** (*bool*) – The immediate flag

basic_qos(*callback=None*, *prefetch_size=0*, *prefetch_count=0*, *all_channels=False*)

Specify quality of service. This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

Parameters

- **callback** (*method*) – The method to callback for Basic.QosOk response
- **prefetch_size** (*int*) – This field specifies the prefetch window size. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls into other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply. The prefetch-size is ignored if the no-ack option is set.
- **prefetch_count** (*int*) – Specifies a prefetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be

sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The prefetch-count is ignored if the no-ack option is set.

- **all_channels** (*bool*) – Should the QoS apply to all channels

basic_reject(*delivery_tag=None*, *requeue=True*)
Reject an incoming message. This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery-tag** (*int*) – The server-assigned delivery tag
- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

basic_recover(*callback=None*, *requeue=False*)
This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method replaces the asynchronous Recover.

Parameters

- **callback** (*method*) – Method to call when receiving Basic.RecoverOk
- **requeue** (*bool*) – If False, the message will be redelivered to the original recipient. If True, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

close(*reply_code=0*, *reply_text='Normal Shutdown'*)
Will invoke a clean shutdown of the channel with the AMQP Broker.

Parameters

- **reply_code** (*int*) – The reply code to close the channel with
- **reply_text** (*str*) – The reply text to close the channel with

confirm_delivery(*callback=None*, *nowait=False*)
Turn on Confirm mode in the channel. Pass in a callback to be notified by the Broker when a message has been confirmed as received or rejected (Basic.Ack, Basic.Nack) from the broker to the publisher.

For more information see: http://www.rabbitmq.com/extensions.html#confirms

Parameters

- **callback** (*method*) – The callback for delivery confirmations
- **nowait** (*bool*) – Do not send a reply frame (Confirm.SelectOk)

consumer_tags
Property method that returns a list of currently active consumers

Return type list

exchange_bind(*callback=None*, *destination=None*, *source=None*, *routing_key=''*, *nowait=False*, *arguments=None*)
Bind an exchange to another exchange.

Parameters

- **callback** (*method*) – The method to call on Exchange.BindOk
- **destination** (*str|unicode*) – The destination exchange to bind
- **source** (*str|unicode*) – The source exchange to bind to
- **routing_key** (*str|unicode*) – The routing key to bind on

- **nowait** (*bool*) – Do not wait for an Exchange.BindOk
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

exchange_declare(*callback=None*, *exchange=None*, *exchange_type='direct'*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *nowait=False*, *arguments=None*)

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected

class.

If passive set, the server will reply with Declare-Ok if the exchange already exists with the same name, and raise an error if not and if the exchange does not already exist, the server MUST raise a channel exception with reply code 404 (not found).

Parameters

- **callback** (*method*) – Call this method on Exchange.DeclareOk
- **exchange** (*str|unicode*) – The exchange name consists of a non-empty sequence of these characters: letters, digits, hyphen, underscore, period, or colon.
- **exchange_type** (*str*) – The exchange type to use
- **passive** (*bool*) – Perform a declare or just check to see if it exists
- **durable** (*bool*) – Survive a reboot of RabbitMQ
- **auto_delete** (*bool*) – Remove when no more queues are bound to it
- **internal** (*bool*) – Can only be published to by other exchanges
- **nowait** (*bool*) – Do not expect an Exchange.DeclareOk response
- **arguments** (*dict*) – Custom key/value pair arguments for the exchange

exchange_delete(*callback=None*, *exchange=None*, *if_unused=False*, *nowait=False*)

Delete the exchange.

Parameters

- **callback** (*method*) – The method to call on Exchange.DeleteOk
- **exchange** (*str|unicode*) – The exchange name
- **if_unused** (*bool*) – only delete if the exchange is unused
- **nowait** (*bool*) – Do not wait for an Exchange.DeleteOk

exchange_unbind(*callback=None*, *destination=None*, *source=None*, *routing_key=''*, *nowait=False*, *arguments=None*)

Unbind an exchange from another exchange.

Parameters

- **callback** (*method*) – The method to call on Exchange.UnbindOk
- **destination** (*str|unicode*) – The destination exchange to unbind
- **source** (*str|unicode*) – The source exchange to unbind from
- **routing_key** (*str|unicode*) – The routing key to unbind
- **nowait** (*bool*) – Do not wait for an Exchange.UnbindOk
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

flow(*callback*, *active*)
Turn Channel flow control off and on. Pass a callback to be notified of the response from the server. active is a bool. Callback should expect a bool in response indicating channel flow state. For more information, please reference:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#channel.flow

Parameters

- **callback** (*method*) – The callback method
- **active** (*bool*) – Turn flow on or off

is_closed
Returns True if the channel is closed.

Return type bool

is_closing
Returns True if the channel is closing.

Return type bool

is_open
Returns True if the channel is open.

Return type bool

open()
Open the channel

queue_bind(*callback*, *queue*, *exchange*, *routing_key*, *nowait=False*, *arguments=None*)
Bind the queue to the specified exchange

Parameters

- **callback** (*method*) – The method to call on Queue.BindOk
- **queue** (*str|unicode*) – The queue to bind to the exchange
- **exchange** (*str|unicode*) – The source exchange to bind to
- **routing_key** (*str|unicode*) – The routing key to bind on
- **nowait** (*bool*) – Do not wait for a Queue.BindOk
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

queue_declare(*callback*, *queue*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=False*, *nowait=False*, *arguments=None*)
Declare queue, create if needed. This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

Parameters

- **callback** (*method*) – The method to call on Queue.DeclareOk
- **queue** (*str|unicode*) – The queue name
- **passive** (*bool*) – Only check to see if the queue exists
- **durable** (*bool*) – Survive reboots of the broker
- **exclusive** (*bool*) – Only allow access by the current connection
- **auto_delete** (*bool*) – Delete after consumer cancels or disconnects

- **nowait** (*bool*) – Do not wait for a Queue.DeclareOk
- **arguments** (*dict*) – Custom key/value arguments for the queue

queue_delete(*callback=None*, *queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*)
Delete a queue from the broker.

Parameters

- **callback** (*method*) – The method to call on Queue.DeleteOk
- **queue** (*str|unicode*) – The queue to delete
- **if_unused** (*bool*) – only delete if it's unused
- **if_empty** (*bool*) – only delete the queue is empty
- **nowait** (*bool*) – Do not wait for a Queue.DeleteOk

queue_purge(*callback=None*, *queue=''*, *nowait=False*)
Purge all of the messages from the specified queue

Parameters

- **callback** (*method*) – The method to call on Queue.PurgeOk
- **str|unicode** – The queue to purge
- **nowait** (*bool*) – Do not expect a Queue.PurgeOk response

queue_unbind(*callback=None*, *queue=''*, *exchange=None*, *routing_key=''*, *arguments=None*)
Unbind a queue from an exchange.

Parameters

- **callback** (*method*) – The method to call on Queue.UnbindOk
- **queue** (*str|unicode*) – The queue to unbind from the exchange
- **exchange** (*str|unicode*) – The source exchange to bind from
- **routing_key** (*str|unicode*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

tx_commit(*callback=None*)
Commit a transaction

Parameters **callback** (*method*) – The callback for delivery confirmations

tx_rollback(*callback=None*)
Rollback a transaction.

Parameters **callback** (*method*) – The callback for delivery confirmations

tx_select(*callback=None*)
Select standard transaction mode. This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

Parameters **callback** (*method*) – The callback for delivery confirmations

_add_callbacks()
Callbacks that add the required behavior for a channel when connecting and connected to a server.

_add_pending_msg(*consumer_tag*, *method_frame*, *header_frame*, *body*)
Add the received message to the pending message stack.

Parameters

- **consumer_tag** (*str*) – The consumer tag for the message
- **method_frame** (pika.frame.Method) – The received method frame
- **header_frame** (pika.frame.Header) – The received header frame
- **body** (*str|unicode*) – The message body

_cleanup()
Remove any callbacks for the channel.

_get_pending_msg(*consumer_tag*)
Get a pending message for the consumer tag from the stack.

Parameters **consumer_tag** (*str*) – The consumer tag to get a message from

Return type tuple(pika.frame.Header, pika.frame.Method, str|unicode)

_handle_content_frame(*frame_value*)
This is invoked by the connection when frames that are not registered with the CallbackManager have been found. This should only be the case when the frames are related to content delivery.

The frame_dispatcher will be invoked which will return the fully formed message in three parts when all of the body frames have been received.

Parameters **frame_value** (*pika.amqp_object.Frame*) – The frame to deliver

_has_content(*method_frame*)
Return a bool if it's a content method as defined by the spec

Parameters **method_frame** (pika.amqp_object.Method) – The method frame received

_on_cancel(*method_frame*)
When the broker cancels a consumer, delete it from our internal dictionary.

Parameters **method_frame** (pika.frame.Method) – The method frame received

_on_cancelok(*method_frame*)

Called in response to a frame from the Broker when the client sends Basic.Cancel

Parameters **method_frame** (pika.frame.Method) – The method frame received

_on_close(*method_frame*)
Handle the case where our channel has been closed for us

Parameters **method_frame** (pika.frame.Method) – The close frame

_on_deliver(*method_frame*, *header_frame*, *body*)
Cope with reentrancy. If a particular consumer is still active when another delivery appears for it, queue the deliveries up until it finally exits.

Parameters

- **method_frame** (pika.frame.Method) – The method frame received
- **header_frame** (pika.frame.Header) – The header frame received
- **body** (*str*) – The body received

_on_eventok(*method_frame*)
Generic events that returned ok that may have internal callbacks. We keep a list of what we've yet to implement so that we don't silently drain events that we don't support.

Parameters **method_frame** (pika.frame.Method) – The method frame received

_on_flow(*method_frame_unused*)
Called if the server sends a Channel.Flow frame.

Parameters **method_frame_unused** (pika.frame.Method) – The Channel.Flow frame

_on_flowok(*method_frame*)
Called in response to us asking the server to toggle on Channel.Flow

Parameters **method_frame** (pika.frame.Method) – The method frame received

_on_getempty(*method_frame*)
When we receive an empty reply do nothing but log it

Parameters **method_frame** (pika.frame.Method) – The method frame received

_on_getok(*method_frame*, *header_frame*, *body*)
Called in reply to a Basic.Get when there is a message.

Parameters

- **method_frame** (pika.frame.Method) – The method frame received
- **header_frame** (pika.frame.Header) – The header frame received
- **body** (*str*) – The body received

_on_openok(*frame_unused*)
Called by our callback handler when we receive a Channel.OpenOk and subsequently calls our _on_openok_callback which was passed into the Channel constructor. The reason we do this is because we want to make sure that the on_open_callback parameter passed into the Channel constructor is not the first callback we make.

Parameters **frame_unused** (pika.frame.Method) – Unused Channel.OpenOk frame

_on_return(*method_frame*, *header_frame*, *body*)
Called if the server sends a Basic.Return frame.

Parameters

- **method_frame** (pika.frame.Method) – The Basic.Return frame
- **header_frame** (pika.frame.Header) – The content header frame
- **body** (*str|unicode*) – The message body

_on_selectok(*method_frame*)
Called when the broker sends a Confirm.SelectOk frame

Parameters **method_frame** (pika.frame.Method) – The method frame received

_on_synchronous_complete(*method_frame_unused*)
This is called when a synchronous command is completed. It will undo the blocking state and send all the frames that stacked up while we were in the blocking state.

Parameters **method_frame_unused** (pika.frame.Method) – The method frame received

_rpc(*method_frame*, *callback=None*, *acceptable_replies=None*)
Shortcut wrapper to the Connection's rpc command using its callback stack, passing in our channel number.

Parameters

- **method_frame** (pika.amqp_object.Method) – The method frame to call
- **callback** (*method*) – The callback for the RPC response

- **acceptable_replies** (*list*) – The replies this RPC call expects

_send_method(*method_frame*, *content=None*)
Shortcut wrapper to send a method through our connection, passing in the channel number

Parameters

- **method_frame** (*pika.object.Method*) – The method frame to send
- **content** (*tuple*) – If set, is a content frame, is tuple of properties and body.

_set_state(*connection_state*)
Set the channel connection state to the specified state value.

Parameters **connection_state** (*int*) – The connection_state value

_shutdown()
Called when close() is invoked either directly or when all of the consumers have been cancelled.

_validate_channel_and_callback(*callback*)

4.4.2 ContentFrameDispatcher

class pika.channel.**ContentFrameDispatcher**
Handle content related frames, building a message and return the message back in three parts upon receipt.

process(*frame_value*)
Invoked by the Channel object when passed frames that are not setup in the rpc process and that don't have explicit reply types defined. This includes Basic.Publish, Basic.GetOk and Basic.Return

Parameters **frame_value** (*Method|Header|Body*) – The frame to process

_finish()
Invoked when all of the message has been received

Return type tuple(pika.frame.Method, pika.frame.Header, str|unicode)

_handle_body_frame(*body_frame*)
Receive body frames and append them to the stack. When the body size matches, call the finish method.

Parameters **body_frame** (Body) – The body frame

Raises pika.exceptions.BodyTooLongError

Return type tuple(pika.frame.Method, pika.frame.Header, str|unicode)|None

_reset()
Reset the values for processing frames

4.5 connection

Note: The following class level documentation is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself. For an overview of how to use adapters, please reference the Connecting to RabbitMQ documentation.

Core connection objects

4.5.1 Parameters

class pika.connection.**Parameters**

Base connection parameters class definition

Parameters

- **DEFAULT_HOST** (*str*) – 'localhost'
- **DEFAULT_PORT** (*int*) – 5672
- **DEFAULT_VIRTUAL_HOST** (*str*) – '/'
- **DEFAULT_USERNAME** (*str*) – 'guest'
- **DEFAULT_PASSWORD** (*str*) – 'guest'
- **DEFAULT_HEARTBEAT_INTERVAL** (*int*) – 0
- **DEFAULT_CHANNEL_MAX** (*int*) – 0
- **DEFAULT_FRAME_MAX** (*int*) – pika.spec.FRAME_MAX_SIZE
- **DEFAULT_LOCALE** (*str*) – 'en_US'
- **DEFAULT_CONNECTION_ATTEMPTS** (*int*) – 1
- **DEFAULT_RETRY_DELAY** (*int | float*) – 2.0
- **DEFAULT_SOCKET_TIMEOUT** (*int | float*) – 0.25
- **DEFAULT_SSL** (*bool*) – False
- **DEFAULT_SSL_OPTIONS** (*dict*) – {}
- **DEFAULT_SSL_PORT** (*int*) – 5671
- **DEFAULT_BACKPRESSURE_DETECTION** (*bool*) – False

DEFAULT_BACKPRESSURE_DETECTION = False

DEFAULT_CONNECTION_ATTEMPTS = 1

DEFAULT_CHANNEL_MAX = 0

DEFAULT_FRAME_MAX = 131072

DEFAULT_HEARTBEAT_INTERVAL = 0

DEFAULT_HOST = 'localhost'

DEFAULT_LOCALE = 'en_US'

DEFAULT_PASSWORD = 'guest'

DEFAULT_PORT = 5672

DEFAULT_RETRY_DELAY = 2.0

DEFAULT_SOCKET_TIMEOUT = 0.25

DEFAULT_SSL = False

DEFAULT_SSL_OPTIONS = {}

DEFAULT_SSL_PORT = 5671

DEFAULT_USERNAME = 'guest'

DEFAULT_VIRTUAL_HOST = '/'

_credentials(*username*, *password*)
Return a plain credentials object for the specified username and password.

Parameters

- **username** (*str*) – The username to use
- **password** (*str*) – The password to use

Return type pika_credentials.PlainCredentials

_validate_backpressure(*backpressure_detection*)
Validate that the backpressure detection option is a bool.

Parameters **backpressure_detection** (*bool*) – The backpressure detection value

Return type bool

Raises TypeError

_validate_channel_max(*channel_max*)
Validate that the channel_max value is an int

Parameters **channel_max** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_connection_attempts(*connection_attempts*)
Validate that the channel_max value is an int

Parameters **connection_attempts** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_credentials(*credentials*)
Validate the credentials passed in are using a valid object type.

Parameters **credentials** (*pika.credentials.Credentials*) – Credentials to validate

Return type bool

Raises TypeError

_validate_frame_max(*frame_max*)

Validate that the frame_max value is an int and does not exceed the maximum frame size and is not less than the frame min size.

Parameters **frame_max** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises InvalidMinimumFrameSize

_validate_heartbeat_interval(*heartbeat_interval*)
Validate that the heartbeat_interval value is an int

Parameters **heartbeat_interval** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_host(*host*)
Validate that the host value is an str

Parameters **host** (*str*) – The value to validate

Return type bool

Raises TypeError

_validate_locale(*locale*)
Validate that the locale value is an str

Parameters **locale** (*str*) – The value to validate

Return type bool

Raises TypeError

_validate_port(*port*)
Validate that the port value is an int

Parameters **port** (*int*) – The value to validate

Return type bool

Raises TypeError

_validate_retry_delay(*retry_delay*)
Validate that the retry_delay value is an int or float

Parameters **retry_delay** (*int | float*) – The value to validate

Return type bool

Raises TypeError

_validate_socket_timeout(*socket_timeout*)
Validate that the socket_timeout value is an int or float

Parameters **socket_timeout** (*int | float*) – The value to validate

Return type bool

Raises TypeError

_validate_ssl(*ssl*)
Validate the SSL toggle is a bool

Parameters **ssl** (*bool*) – The SSL enabled/disabled value

Return type bool

Raises TypeError

_validate_ssl_options(*ssl_options*)
Validate the SSL options value is a dictionary.

Parameters **ssl_options** (*dict | None*) – SSL Options to validate

Return type bool

Raises TypeError

_validate_virtual_host(*virtual_host*)
Validate that the virtual_host value is an str

Parameters **virtual_host** (*str*) – The value to validate

Return type bool

Raises TypeError

4.5.2 ConnectionParameters

class pika.connection.**ConnectionParameters**(*host=None*, *port=None*, *virtual_host=None*, *credentials=None*, *channel_max=None*, *frame_max=None*, *heartbeat_interval=None*, *ssl=None*, *ssl_options=None*, *connection_attempts=None*, *retry_delay=None*, *socket_timeout=None*, *locale=None*, *backpressure_detection=None*)
Connection parameters object that is passed into the connection adapter upon construction.

DEFAULT_BACKPRESSURE_DETECTION = False

DEFAULT_CHANNEL_MAX = 0

DEFAULT_CONNECTION_ATTEMPTS = 1

DEFAULT_FRAME_MAX = 131072

DEFAULT_HEARTBEAT_INTERVAL = 0

DEFAULT_HOST = 'localhost'

DEFAULT_LOCALE = 'en_US'

DEFAULT_PASSWORD = 'guest'

DEFAULT_PORT = 5672

DEFAULT_RETRY_DELAY = 2.0

DEFAULT_SOCKET_TIMEOUT = 0.25

DEFAULT_SSL = False

DEFAULT_SSL_OPTIONS = {}

DEFAULT_SSL_PORT = 5671

DEFAULT_USERNAME = 'guest'

DEFAULT_VIRTUAL_HOST = '/'

_credentials(*username*, *password*)
Return a plain credentials object for the specified username and password.

Parameters

- **username** (*str*) – The username to use
- **password** (*str*) – The password to use

Return type pika_credentials.PlainCredentials

_validate_backpressure(*backpressure_detection*)
Validate that the backpressure detection option is a bool.

Parameters **backpressure_detection** (*bool*) – The backpressure detection value

Return type bool

Raises TypeError

_validate_channel_max(*channel_max*)
Validate that the channel_max value is an int

Parameters **channel_max** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_connection_attempts(*connection_attempts*)
Validate that the channel_max value is an int

Parameters **connection_attempts** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_credentials(*credentials*)
Validate the credentials passed in are using a valid object type.

Parameters **credentials** (*pika.credentials.Credentials*) – Credentials to validate

Return type bool

Raises TypeError

_validate_frame_max(*frame_max*)

Validate that the frame_max value is an int and does not exceed the maximum frame size and is not less than the frame min size.

Parameters **frame_max** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises InvalidMinimumFrameSize

_validate_heartbeat_interval(*heartbeat_interval*)
Validate that the heartbeat_interval value is an int

Parameters **heartbeat_interval** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_host(*host*)
Validate that the host value is an str

Parameters **host** (*str*) – The value to validate

Return type bool

Raises TypeError

_validate_locale(*locale*)
Validate that the locale value is an str

Parameters **locale** (*str*) – The value to validate

Return type bool

Raises TypeError

_validate_port(*port*)
Validate that the port value is an int

Parameters **port** (*int*) – The value to validate

Return type bool

Raises TypeError

_validate_retry_delay(*retry_delay*)
Validate that the retry_delay value is an int or float

Parameters **retry_delay** (*int | float*) – The value to validate

Return type bool

Raises TypeError

_validate_socket_timeout(*socket_timeout*)
Validate that the socket_timeout value is an int or float

Parameters **socket_timeout** (*int | float*) – The value to validate

Return type bool

Raises TypeError

_validate_ssl(*ssl*)
Validate the SSL toggle is a bool

Parameters **ssl** (*bool*) – The SSL enabled/disabled value

Return type bool

Raises TypeError

_validate_ssl_options(*ssl_options*)
Validate the SSL options value is a dictionary.

Parameters **ssl_options** (*dict | None*) – SSL Options to validate

Return type bool

Raises TypeError

_validate_virtual_host(*virtual_host*)
Validate that the virtual_host value is an str

Parameters **virtual_host** (*str*) – The value to validate

Return type bool

Raises TypeError

4.5.3 URLParameters

class pika.connection.**URLParameters**(*url*)

Create a Connection parameters object based off of URIParameters

_process_url(*url*)

Take an AMQP URL and break it up into the various parameters.

Parameters **url** (*str*) – The URL to parse

DEFAULT_BACKPRESSURE_DETECTION = False

DEFAULT_CHANNEL_MAX = 0

DEFAULT_CONNECTION_ATTEMPTS = 1

DEFAULT_FRAME_MAX = 131072

DEFAULT_HEARTBEAT_INTERVAL = 0

DEFAULT_HOST = 'localhost'

DEFAULT_LOCALE = 'en_US'

DEFAULT_PASSWORD = 'guest'

DEFAULT_PORT = 5672

DEFAULT_RETRY_DELAY = 2.0

DEFAULT_SOCKET_TIMEOUT = 0.25

DEFAULT_SSL = False

DEFAULT_SSL_OPTIONS = {}

DEFAULT_SSL_PORT = 5671

DEFAULT_USERNAME = 'guest'

DEFAULT_VIRTUAL_HOST = '/'

_credentials(*username*, *password*)

Return a plain credentials object for the specified username and password.

Parameters

- **username** (*str*) – The username to use
- **password** (*str*) – The password to use

Return type pika_credentials.PlainCredentials

_validate_backpressure(*backpressure_detection*)

Validate that the backpressure detection option is a bool.

Parameters **backpressure_detection** (*bool*) – The backpressure detection value

Return type bool

Raises TypeError

_validate_channel_max(*channel_max*)

Validate that the channel_max value is an int

Parameters **channel_max** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_connection_attempts(*connection_attempts*)
Validate that the channel_max value is an int

Parameters **connection_attempts** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_credentials(*credentials*)
Validate the credentials passed in are using a valid object type.

Parameters **credentials** (*pika.credentials.Credentials*) – Credentials to validate

Return type bool

Raises TypeError

_validate_frame_max(*frame_max*)

Validate that the frame_max value is an int and does not exceed the maximum frame size and is not less than the frame min size.

Parameters **frame_max** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises InvalidMinimumFrameSize

_validate_heartbeat_interval(*heartbeat_interval*)
Validate that the heartbeat_interval value is an int

Parameters **heartbeat_interval** (*int*) – The value to validate

Return type bool

Raises TypeError

Raises ValueError

_validate_host(*host*)
Validate that the host value is an str

Parameters **host** (*str*) – The value to validate

Return type bool

Raises TypeError

_validate_locale(*locale*)
Validate that the locale value is an str

Parameters **locale** (*str*) – The value to validate

Return type bool

Raises TypeError

_validate_port(*port*)
Validate that the port value is an int

Parameters **port** (*int*) – The value to validate

Return type bool

Raises TypeError

_validate_retry_delay(*retry_delay*)
Validate that the retry_delay value is an int or float

Parameters **retry_delay** (*int | float*) – The value to validate

Return type bool

Raises TypeError

_validate_socket_timeout(*socket_timeout*)
Validate that the socket_timeout value is an int or float

Parameters **socket_timeout** (*int | float*) – The value to validate

Return type bool

Raises TypeError

_validate_ssl(*ssl*)
Validate the SSL toggle is a bool

Parameters **ssl** (*bool*) – The SSL enabled/disabled value

Return type bool

Raises TypeError

_validate_ssl_options(*ssl_options*)
Validate the SSL options value is a dictionary.

Parameters **ssl_options** (*dict | None*) – SSL Options to validate

Return type bool

Raises TypeError

_validate_virtual_host(*virtual_host*)
Validate that the virtual_host value is an str

Parameters **virtual_host** (*str*) – The value to validate

Return type bool

Raises TypeError

4.5.4 Connection

class pika.connection.**Connection**(*parameters=None*, *on_open_callback=None*)
This is the core class that implements communication with RabbitMQ. This class should not be invoked directly but rather through the use of an adapter such as SelectConnection or BlockingConnection.

CONNECTION_CLOSED = 0

CONNECTION_INIT = 1

CONNECTION_PROTOCOL = 2

CONNECTION_START = 3

CONNECTION_TUNE = 4

CONNECTION_OPEN = 5

CONNECTION_CLOSING = 6

add_backpressure_callback(*callback_method*)
Call method "callback" when pika believes backpressure is being applied.

Parameters **callback_method** (*method*) – The method to call

add_on_close_callback(*callback_method*)
Add a callback notification when the connection has closed.

Parameters **callback_method** (*method*) – The callback when the channel is opened

add_on_open_callback(*callback_method*)
Add a callback notification when the connection has opened.

Parameters **callback_method** (*method*) – The callback when the channel is opened

add_timeout(*deadline*, *callback_method*)
Adapters should override to call the callback after the specified number of seconds have elapsed, using a timer, or a thread, or similar.

Parameters

- **deadline** (*int*) – The number of seconds to wait to call callback
- **callback_method** (*method*) – The callback method

channel(*on_open_callback*, *channel_number=None*)
Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters

- **on_open_callback** (*method*) – The callback when the channel is opened
- **channel_number** (*int*) – The channel number to use, defaults to the next available.

close(*reply_code=200*, *reply_text='Normal shutdown'*)
Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

remove_timeout(*callback_method*)
Adapters should override to call the callback after the specified number of seconds have elapsed, using a timer, or a thread, or similar.

Parameters **callback_method** (*method*) – The callback to remove a timeout for

set_backpressure_multiplier(*value=10*)
Alter the backpressure multiplier value. We set this to 10 by default. This value is used to raise warnings and trigger the backpressure callback.

Parameters **value** (*int*) – The multiplier value to set

is_closed
Returns a boolean reporting the current connection state.

is_closing
Returns a boolean reporting the current connection state.

is_open
Returns a boolean reporting the current connection state.

basic_nack
Specifies if the server supports basic.nack on the active connection.

Return type bool

consumer_cancel_notify
Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

exchange_exchange_bindings
Specifies if the active connection supports exchange to exchange bindings.

Return type bool

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

_adapter_connect()
Subclasses should override to set up the outbound socket connection.

Raises NotImplementedError

_adapter_disconnect()
Subclasses should override this to cause the underlying transport (socket) to close.

Raises NotImplementedError

_add_channel_callbacks(*channel_number*)
Add the appropriate callbacks for the specified channel number.

Parameters **channel_number** (*int*) – The channel number for the callbacks

_add_connection_start_callback()
Add a callback for when a Connection.Start frame is received from the broker.

_add_connection_tune_callback()
Add a callback for when a Connection.Tune frame is received.

_append_frame_buffer(*bytes*)
Append the bytes to the frame buffer.

Parameters **bytes** (*str*) – The bytes to append to the frame buffer

_buffer_size
Return the suggested buffer size from the connection state/tune or the default if that is None.

Return type int

_check_for_protocol_mismatch(*value*)
Invoked when starting a connection to make sure it's a supported protocol.

Parameters **value** (pika.frame.Method) – The frame to check

Raises ProtocolVersionMismatch

_client_properties
Return the client properties dictionary.

Return type dict

_close_channels(*reply_code*, *reply_text*)
Close the open channels with the specified reply_code and reply_text.

Parameters

- **reply_code** (*int*) – The code for why the channels are being closed
- **reply_text** (*str*) – The text reason for why the channels are closing

_combine(*a*, *b*)
Pass in two values, if a is 0, return b otherwise if b is 0, return a. If neither case matches return the smallest value.

Parameters

- **a** (*int*) – The first value
- **b** (*int*) – The second value

Return type int

_connect()
Call the Adapter's connect method after letting the ReconnectionStrategy know.

_create_channel(*channel_number*, *on_open_callback*)
Create a new channel using the specified channel number and calling back the method specified by on_open_callback

Parameters

- **channel_number** (*int*) – The channel number to use
- **on_open_callback** (*method*) – The callback when the channel is opened

_create_heartbeat_checker()
Create a heartbeat checker instance if there is a heartbeat interval set.

Return type pika.heartbeat.Heartbeat

_deliver_frame_to_channel(*value*)
Deliver the frame to the channel specified in the frame.

Parameters **value** (pika.frame.Method) – The frame to deliver

_detect_backpressure()
Attempt to calculate if TCP backpressure is being applied due to our outbound buffer being larger than the average frame size over a window of frames.

_ensure_closed()
If the connection is not closed, close it.

_flush_outbound()
Adapters should override to flush the contents of outbound_buffer out along the socket.

Raises NotImplementedError

_get_body_frame_max_length()
Calculate the maximum amount of bytes that can be in a body frame.

Return type int

_get_credentials(*method_frame*)
Get credentials for authentication.

Parameters **method_frame** (*pika.frame.MethodFrame*) – The Connection.Start frame

Return type tuple(str, str)

_has_open_channels
Returns true if channels are open.

Return type bool

_has_pending_callbacks(*value*)
Return true if there are any callbacks pending for the specified frame.

Parameters **value** (pika.frame.Method) – The frame to check

Return type bool

_init_connection_state()
Initialize or reset all of the internal state variables for a given connection. On disconnect or reconnect all of the state needs to be wiped.

_is_basic_deliver_frame(*frame_value*)
Returns true if the frame is a Basic.Deliver

Parameters **frame_value** (pika.frame.Method) – The frame to check

Return type bool

_is_connection_close_frame(*value*)
Returns true if the frame is a Connection.Close frame.

Parameters **value** (pika.frame.Method) – The frame to check

Return type bool

_is_method_frame(*value*)
Returns true if the frame is a method frame.

Parameters **value** (pika.frame.Frame) – The frame to evaluate

Return type bool

_is_protocol_header_frame(*value*)
Returns True if it's a protocol header frame.

Return type bool

_next_channel_number()
Return the next available channel number or raise on exception.

Return type int

_on_channel_closeok(*method_frame*)
Remove the channel from the dict of channels when Channel.CloseOk is sent.

Parameters **method_frame** (spec.Channel.CloseOk) – The response

_on_close_ready()
Called when the Connection is in a state that it can close after a close has been requested. This happens, for example, when all of the channels are closed that were open when the close request was made.

_on_connected()
This is called by our connection Adapter to let us know that we've connected and we can notify our connection strategy.

_on_connection_closed(*method_frame*, *from_adapter=False*)
Called when the connection is closed remotely. The from_adapter value will be true if the connection adapter has been disconnected from the broker and the method was invoked directly instead of by receiving a Connection.Close frame.

Parameters

- **pika.frame.Method** – The Connection.Close frame
- **from_adapter** (*bool*) – Called by the connection adapter

_on_connection_open(*method_frame*)
This is called once we have tuned the connection with the server and called the Connection.Open on the server and it has replied with Connection.Ok.

_on_connection_start(*method_frame*)
This is called as a callback once we have received a Connection.Start from the server.

Parameters **method_frame** (pika.frame.Method) – The frame received

Raises UnexpectedFrameError

_on_connection_tune(*method_frame*)
Once the Broker sends back a Connection.Tune, we will set our tuning variables that have been returned to us and kick off the Heartbeat monitor if required, send our TuneOk and then the Connection. Open rpc call on channel 0.

Parameters **method_frame** (pika.frame.Method) – The frame received

_on_data_available(*data_in*)
This is called by our Adapter, passing in the data from the socket. As long as we have buffer try and map out frame data.

Parameters **data_in** (*str*) – The data that is available to read

_process_callbacks(*frame_value*)
Process the callbacks for the frame if the frame is a method frame and if it has any callbacks pending.

Parameters **frame_value** (pika.frame.Method) – The frame to process

Return type bool

_process_connection_closed_callbacks()
Process any callbacks that should be called when the connection is closed.

_process_frame(*frame_value*)
Process an inbound frame from the socket.

Parameters **frame_value** (*pika.frame.Frame | pika.frame.Method*) – The frame to process

_read_frame()
Try and read from the frame buffer and decode a frame.

Rtype tuple (int, pika.frame.Frame)

_reject_out_of_band_delivery(*channel_number*, *delivery_tag*)
Reject a delivery on the specified channel number and delivery tag because said channel no longer exists.

Parameters

- **channel_number** (*int*) – The channel number
- **delivery_tag** (*int*) – The delivery tag

_remove_callback(*channel_number*, *method_frame*)
Remove the specified method_frame callback if it is set for the specified channel number.

Parameters

- **channel_number** (*int*) – The channel number to remove the callback on
- **pika.object.Method** – The method frame for the callback

_remove_callbacks(*channel_number*, *method_frames*)
Remove the callbacks for the specified channel number and list of method frames.

Parameters

- **channel_number** (*int*) – The channel number to remove the callback on
- **method_frames** (*list*) – The method frames for the callback

_remove_connection_callbacks()
Remove all callbacks for the connection

_rpc(*channel_number*, *method_frame*, *callback_method=None*, *acceptable_replies=None*)
Make an RPC call for the given callback, channel number and method. acceptable_replies lists out what responses we'll process from the server with the specified callback.

Parameters

- **channel_number** (*int*) – The channel number for the RPC call
- **method_frame** (*pika.object.Method*) – The method frame to call
- **callback_method** (*method*) – The callback for the RPC response
- **acceptable_replies** (*list*) – The replies this RPC call expects

_send_connection_close(*reply_code*, *reply_text*)
Send a Connection.Close method frame.

Parameters

- **reply_code** (*int*) – The reason for the close
- **reply_text** (*str*) – The text reason for the close

_send_connection_open()
Send a Connection.Open frame

_send_connection_start_ok(*authentication_type*, *response*)
Send a Connection.StartOk frame

Parameters

- **authentication_type** (*str*) – The auth type value
- **response** (*str*) – The encoded value to send

_send_connection_tune_ok()
Send a Connection.TuneOk frame

_send_frame(*frame_value*)
This appends the fully generated frame to send to the broker to the output buffer which will be then sent via the connection adapter.

Parameters **frame_value** (*pika.frame.Frame|pika.frame.ProtocolHeader*) – The frame to write

_send_method(*channel_number*, *method_frame*, *content=None*)
Constructs a RPC method frame and then sends it to the broker.

Parameters

- **channel_number** (*int*) – The channel number for the frame
- **method_frame** (*pika.object.Method*) – The method frame to send
- **content** (*tuple*) – If set, is a content frame, is tuple of properties and body.

_set_connection_state(*connection_state*)
Set the connection state.

Parameters **connection_state** (*int*) – The connection state to set

_set_server_information(*method_frame*)
Set the server properties and capabilities

Parameters **method_frame** (*spec.connection.Start*) – The Connection.Start frame

_trim_frame_buffer(*byte_count*)
Trim the leading N bytes off the frame buffer and increment the counter that keeps track of how many bytes have been read/used from the socket.

Parameters **byte_count** (*int*) – The number of bytes consumed

4.6 credentials

Credentials Classes

4.6.1 PlainCredentials

class pika.credentials.**PlainCredentials**(*username*, *password*, *erase_on_connect=False*)
The PlainCredentials class returns the properly formatted username and password to the Connection. As of this version of Pika, only PlainCredentials are supported. To authenticate with Pika, simply create a credentials object passing in the username and password and pass that to the ConnectionParameters object.

If you do not pass in credentials to the ConnectionParameters object, it will create credentials for 'guest' with the password of 'guest'.

If you pass True to erase_on_connect the credentials will not be stored in memory after the Connection attempt has been made.

erase_credentials()
Called by Connection when it no longer needs the credentials

response_for(*start*)
Validate that this type of authentication is supported

Parameters **start** (spec.Connection.Start) – Connection.Start method

Return type tuple(str|None, str|None)

4.6.2 ExternalCredentials

class pika.credentials.**ExternalCredentials**
The ExternalCredentials class allows the connection to use EXTERNAL authentication, generally with a client SSL certificate.

erase_credentials()
Called by Connection when it no longer needs the credentials

response_for(*start*)
Validate that this type of authentication is supported

Parameters **start** (spec.Connection.Start) – Connection.Start method

Return type tuple(str|None, str|None)

4.7 data

The data module is for encoding and decoding of AMQP binary encoded data.

Note: This class is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

AMQP Table Encoding/Decoding

pika.data.**encode_table**(*pieces*, *table*)
Encode a dict as an AMQP table appending the encded table to the pieces list passed in.

Parameters

- **pieces** (*list*) – Already encoded frame pieces
- **table** (*dict*) – The dict to encode

Return type int

pika.data.**encode_value**(*pieces*, *value*)
Encode the value passed in and append it to the pieces list returning the the size of the encoded value.

Parameters

- **pieces** (*list*) – Already encoded values
- **value** (*any*) – The value to encode

Return type int

pika.data.**decode_table**(*encoded*, *offset*)
Decode the AMQP table passed in from the encoded value returning the decoded result and the number of bytes read plus the offset.

Parameters

- **encoded** (*str*) – The binary encoded data to decode
- **offset** (*int*) – The starting byte offset

Return type tuple

pika.data.**decode_value**(*encoded*, *offset*)
Decode the value passed in returning the decoded value and the number of bytes read in addition to the starting offset.

Parameters

- **encoded** (*str*) – The binary encoded data to decode
- **offset** (*int*) – The starting byte offset

Return type tuple

Raises pika.exceptions.InvalidFieldTypeException

4.8 exceptions

Pika specific exceptions

exception pika.exceptions.**AMQPChannelError**

exception pika.exceptions.**AMQPConnectionError**

exception pika.exceptions.**AMQPError**

exception pika.exceptions.**AuthenticationError**

exception pika.exceptions.**BodyTooLongError**

exception pika.exceptions.**ChannelClosed**

exception pika.exceptions.**ChannelError**

exception pika.exceptions.**ConnectionClosed**

exception pika.exceptions.**ConsumerCancelled**

exception pika.exceptions.**DuplicateConsumerTag**

exception pika.exceptions.**IncompatibleProtocolError**

exception pika.exceptions.**InvalidChannelNumber**

exception pika.exceptions.**InvalidFieldTypeException**

exception pika.exceptions.**InvalidFrameError**

exception pika.exceptions.**InvalidMaximumFrameSize**

exception pika.exceptions.**InvalidMinimumFrameSize**

exception pika.exceptions.**MethodNotImplemented**

exception pika.exceptions.**NoFreeChannels**

exception pika.exceptions.**ProbableAccessDeniedError**

exception pika.exceptions.**ProbableAuthenticationError**

exception pika.exceptions.**ProtocolSyntaxError**

exception pika.exceptions.**ProtocolVersionMismatch**

exception pika.exceptions.**UnexpectedFrameError**

exception pika.exceptions.**UnspportedAMQPFieldException**

4.9 frame

The frame module contains the object structure for mapping AMQP Classes and Methods on to Python objects. In addition frame contains the Dispatcher class which steps through the synchronous receipt order of frames received for Basic.Deliver and Basic.GetOk.

Note: This class level documentation is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

Frame objects that do the frame demarshaling and marshaling.

4.9.1 Frame

class pika.frame.**Frame**(*frame_type*, *channel_number*)

Base Frame object mapping. Defines a behavior for all child classes for assignment of core attributes and implementation of the a core _marshal method which child classes use to create the binary AMQP frame.

_marshal(*pieces*)

Create the full AMQP wire protocol frame data representation

Return type str

marshal()

To be ended by child classes

:raises NotImplementedError

4.9.2 Method

class pika.frame.**Method**(*channel_number*, *method*)

Base Method frame object mapping. AMQP method frames are mapped on top of this class for creating or accessing their data and attributes.

marshal()

Return the AMQP binary encoded value of the frame

Return type str

4.9.3 Header

class pika.frame.**Header**(*channel_number*, *body_size*, *props*)

Header frame object mapping. AMQP content header frames are mapped on top of this class for creating or accessing their data and attributes.

marshal()

Return the AMQP binary encoded value of the frame

Return type str

4.9.4 Body

class pika.frame.**Body**(*channel_number*, *fragment*)

Body frame object mapping class. AMQP content body frames are mapped on to this base class for getting/setting of attributes/data.

marshal()

Return the AMQP binary encoded value of the frame

Return type str

4.9.5 ProtocolHeader

class pika.frame.**ProtocolHeader**(*major=None*, *minor=None*, *revision=None*)
AMQP Protocol header frame class which provides a pythonic interface for creating AMQP Protocol headers

marshal()
Return the full AMQP wire protocol frame data representation of the ProtocolHeader frame

Return type str

4.9.6 Frame Decoding

pika.frame.**decode_frame**(*data_in*)
Receives raw socket data and attempts to turn it into a frame. Returns bytes used to make the frame and the frame

Parameters **data_in** (*str*) – Theraw data stream

Return type tuple(bytes consumed, frame)

Raises pika.exceptions.InvalidFrameError

4.10 heartbeat

Note: This class is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

Handle AMQP Heartbeats

4.10.1 HeartbeatChecker

class pika.heartbeat.**HeartbeatChecker**(*connection*, *interval*, *idle_count=2*)
Checks to make sure that our heartbeat is received at the expected intervals.

MAX_IDLE_COUNT = 2

_CONNECTION_FORCED = 320

_STALE_CONNECTION = 'Too Many Missed Heartbeats, No reply in %i seconds'

active
Return True if the connection's heartbeat attribute is set to this instance.

:rtype True

bytes_received_on_connection
Return the number of bytes received by the connection bytes object.

:rtype int

connection_is_idle
Returns true if the byte count hasn't changed in enough intervals to trip the max idle threshold.

received()
Called when a heartbeat is received

send_and_check()
Invoked by a timer to send a heartbeat when we need to, check to see if we've missed any heartbeats and disconnect our connection if it's been idle too long.

_close_connection()
Close the connection with the AMQP Connection-Forced value.

_has_received_data
Returns True if the connection has received data on the connection.

Return type bool

_new_heartbeat_frame()
Return a new heartbeat frame.

:rtype pika.frame.Heartbeat

_send_heartbeat_frame()
Send a heartbeat frame on the connection.

_setup_timer()
Use the connection objects delayed_call function which is implemented by the Adapter for calling the check_heartbeats function every interval seconds.

_start_timer()
If the connection still has this object set for heartbeats, add a new timer.

_update_counters()
Update the internal counters for bytes sent and received and the number of frames received

4.11 simplebuffer

Note: This class is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

4.11.1 SimpleBuffer

This is an implementation of a simple buffer. SimpleBuffer just handles a string of bytes. The clue, is that you can pop data from the beginning and append data to the end.

It's ideal to use as a network buffer, from which you send data to the socket. Use this to avoid concatenating or splitting large strings.

class pika.simplebuffer.**SimpleBuffer**(*data=None*)
A simple buffer that will handle storing, reading and sending strings to a socket.

```
>>> b = SimpleBuffer("abcdef")
>>> b.read_and_consume(3)
'abc'
>>> b.write(None, '')
>>> b.read(0)
''
>>> repr(b)
"<SimpleBuffer of 3 bytes, 6 total size, 'def'>"
>>> str(b)
"<SimpleBuffer of 3 bytes, 6 total size, 'def'>"
```

```
>>> b.flush()
>>> b.read(1)
''
```

write(**data_strings*)
Append given strings to the buffer.

Parameters **data_strings** (*str|unicode*) – Value to write to the buffer

read(*size=None*)
Read the data from the buffer, at most 'size' bytes.

Parameters **size** (*int*) – The number of bytes to read

consume(*size*)
Move pointer and discard first 'size' bytes.

Parameters **size** (*int*) – The number of bytes to consume

read_and_consume(*size*)
Read up to 'size' bytes, also remove it from the buffer.

Parameters **size** (*int*) – The number of bytes to read and consume

Return type str

send_to_socket(*sd*)
Faster way of sending buffer data to socket 'sd'.

Parameters **sd** (*socket.socket*) – The socket to send data to

Return type int

flush()
Remove all the data from buffer.

4.12 spec

Note: This class is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

spec.py is an auto-generated python module which is created by the codegen.py application. It contains all of the information about the protocol that is required for Pika to communicate with RabbitMQ.

class pika.spec.**Connection**

INDEX = 10

NAME = 'Connection'

class **Start**(*version_major=0*, *version_minor=9*, *server_properties=None*, *mechanisms='PLAIN'*, *locales='en_US'*)

INDEX = 655370

NAME = 'Connection.Start'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**StartOk**(*client_properties=None*, *mechanism='PLAIN'*, *response=None*, *locale='en_US'*)

INDEX = 655371

NAME = 'Connection.StartOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**Secure**(*challenge=None*)

INDEX = 655380

NAME = 'Connection.Secure'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.

Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**SecureOk**(*response=None*)

INDEX = 655381

NAME = 'Connection.SecureOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**Tune**(*channel_max=0*, *frame_max=0*, *heartbeat=0*)

INDEX = 655390

NAME = 'Connection.Tune'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**TuneOk**(*channel_max=0*, *frame_max=0*, *heartbeat=0*)

INDEX = 655391

NAME = 'Connection.TuneOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**Open**(*virtual_host='/'*, *capabilities=''*, *insist=False*)

INDEX = 655400

NAME = 'Connection.Open'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**OpenOk**(*known_hosts=''*)

INDEX = 655401

NAME = 'Connection.OpenOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters

- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**Close**(*reply_code=None*, *reply_text=''*, *class_id=None*, *method_id=None*)

INDEX = 655410

NAME = 'Connection.Close'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Connection.**CloseOk**

INDEX = 655411

NAME = 'Connection.CloseOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Channel**

INDEX = 20

NAME = 'Channel'

class Open(*out_of_band=''*)

INDEX = 1310730

NAME = 'Channel.Open'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Channel.**OpenOk**(*channel_id=''*)

INDEX = 1310731

NAME = 'Channel.OpenOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Channel.**Flow**(*active=None*)

INDEX = 1310740

NAME = 'Channel.Flow'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Channel.**FlowOk**(*active=None*)

INDEX = 1310741

NAME = 'Channel.FlowOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Channel.**Close**(*reply_code=None*, *reply_text=''*, *class_id=None*, *method_id=None*)

INDEX = 1310760

NAME = 'Channel.Close'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters

- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Channel.**CloseOk**

INDEX = 1310761

NAME = 'Channel.CloseOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Access**

INDEX = 30

NAME = 'Access'

class Request(*realm='/data'*, *exclusive=False*, *passive=True*, *active=True*, *write=True*, *read=True*)

INDEX = 1966090

NAME = 'Access.Request'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Access.**RequestOk**(*ticket=1*)

INDEX = 1966091

NAME = 'Access.RequestOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Exchange**

INDEX = 40

NAME = 'Exchange'

class Declare(*ticket=0*, *exchange=None*, *type='direct'*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *nowait=False*, *arguments={}*)

INDEX = 2621450

NAME = 'Exchange.Declare'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Exchange.**DeclareOk**

INDEX = 2621451

NAME = 'Exchange.DeclareOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Exchange.**Delete**(*ticket=0*, *exchange=None*, *if_unused=False*, *nowait=False*)

INDEX = 2621460

NAME = 'Exchange.Delete'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Exchange.**DeleteOk**

INDEX = 2621461

NAME = 'Exchange.DeleteOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Exchange.**Bind**(*ticket=0*, *destination=None*, *source=None*, *routing_key=''*, *nowait=False*, *arguments={}*)

INDEX = 2621470

NAME = 'Exchange.Bind'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Exchange.**BindOk**

INDEX = 2621471

NAME = 'Exchange.BindOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties

- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Exchange.**Unbind**(*ticket=0*, *destination=None*, *source=None*, *routing_key=''*, *nowait=False*, *arguments={}*)

INDEX = 2621480

NAME = 'Exchange.Unbind'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Exchange.**UnbindOk**

INDEX = 2621491

NAME = 'Exchange.UnbindOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Queue**

INDEX = 50

NAME = 'Queue'

class Declare(*ticket=0*, *queue=''*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=False*, *nowait=False*, *arguments={}*)

INDEX = 3276810

NAME = 'Queue.Declare'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**DeclareOk**(*queue=None*, *message_count=None*, *consumer_count=None*)

INDEX = 3276811

NAME = 'Queue.DeclareOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**Bind**(*ticket=0*, *queue=''*, *exchange=None*, *routing_key=''*, *nowait=False*, *arguments={}*)

INDEX = 3276820

NAME = 'Queue.Bind'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**BindOk**

INDEX = 3276821

NAME = 'Queue.BindOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**Purge**(*ticket=0*, *queue=''*, *nowait=False*)

INDEX = 3276830

NAME = 'Queue.Purge'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**PurgeOk**(*message_count=None*)

INDEX = 3276831

NAME = 'Queue.PurgeOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**Delete**(*ticket=0*, *queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*)

INDEX = 3276840

NAME = 'Queue.Delete'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**DeleteOk**(*message_count=None*)

INDEX = 3276841

NAME = 'Queue.DeleteOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**Unbind**(*ticket=0*, *queue=''*, *exchange=None*, *routing_key=''*, *arguments={}*)

INDEX = 3276850

NAME = 'Queue.Unbind'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Queue.**UnbindOk**

INDEX = 3276851

NAME = 'Queue.UnbindOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Basic**

INDEX = 60

NAME = 'Basic'

class Qos(*prefetch_size=0*, *prefetch_count=0*, *global_=False*)

INDEX = 3932170

NAME = 'Basic.Qos'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**QosOk**

INDEX = 3932171

NAME = 'Basic.QosOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Consume**(*ticket=0*, *queue=''*, *consumer_tag=''*, *no_local=False*, *no_ack=False*, *exclusive=False*, *nowait=False*, *arguments={}*)

INDEX = 3932180

NAME = 'Basic.Consume'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**ConsumeOk**(*consumer_tag=None*)

INDEX = 3932181

NAME = 'Basic.ConsumeOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.

Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Cancel**(*consumer_tag=None*, *nowait=False*)

INDEX = 3932190

NAME = 'Basic.Cancel'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**CancelOk**(*consumer_tag=None*)

INDEX = 3932191

NAME = 'Basic.CancelOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Publish**(*ticket=0*, *exchange=''*, *routing_key=''*, *mandatory=False*, *immediate=False*)

INDEX = 3932200

NAME = 'Basic.Publish'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Return**(*reply_code=None*, *reply_text=''*, *exchange=None*, *routing_key=None*)

INDEX = 3932210

NAME = 'Basic.Return'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Deliver**(*consumer_tag=None*, *delivery_tag=None*, *redelivered=False*, *exchange=None*, *routing_key=None*)

INDEX = 3932220

NAME = 'Basic.Deliver'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters

- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Get**(*ticket=0*, *queue=''*, *no_ack=False*)

INDEX = 3932230

NAME = 'Basic.Get'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**GetOk**(*delivery_tag=None*, *redelivered=False*, *exchange=None*, *routing_key=None*, *message_count=None*)

INDEX = 3932231

NAME = 'Basic.GetOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**GetEmpty**(*cluster_id=''*)

INDEX = 3932232

NAME = 'Basic.GetEmpty'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Ack**(*delivery_tag=0*, *multiple=False*)

INDEX = 3932240

NAME = 'Basic.Ack'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Reject**(*delivery_tag=None*, *requeue=True*)

INDEX = 3932250

NAME = 'Basic.Reject'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**RecoverAsync**(*requeue=False*)

INDEX = 3932260

NAME = 'Basic.RecoverAsync'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Recover**(*requeue=False*)

INDEX = 3932270

NAME = 'Basic.Recover'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.

Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**RecoverOk**

INDEX = 3932271

NAME = 'Basic.RecoverOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Basic.**Nack**(*delivery_tag=0*, *multiple=False*, *requeue=True*)

INDEX = 3932280

NAME = 'Basic.Nack'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Tx**

INDEX = 90

NAME = 'Tx'

class Select

INDEX = 5898250

NAME = 'Tx.Select'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Tx.SelectOk

INDEX = 5898251

NAME = 'Tx.SelectOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Tx.Commit

INDEX = 5898260

NAME = 'Tx.Commit'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Tx.**CommitOk**

INDEX = 5898261

NAME = 'Tx.CommitOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Tx.**Rollback**

INDEX = 5898270

NAME = 'Tx.Rollback'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Tx.**RollbackOk**

INDEX = 5898271

NAME = 'Tx.RollbackOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Confirm**

INDEX = 85

NAME = 'Confirm'

class Select(*nowait=False*)

INDEX = 5570570

NAME = 'Confirm.Select'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Confirm.**SelectOk**

INDEX = 5570571

NAME = 'Confirm.SelectOk'

synchronous

decode(*encoded*, *offset=0*)

encode()

_set_content(*properties*, *body*)
If the method is a content frame, set the properties and body to be carried as attributes of the class.
Parameters
- **properties** (*pika.frame.Properties*) – AMQP Basic Properties
- **body** (*str|unicode*) – The message body

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**BasicProperties**(*content_type=None*, *content_encoding=None*, *headers=None*, *delivery_mode=None*, *priority=None*, *correlation_id=None*, *reply_to=None*, *expiration=None*, *message_id=None*, *timestamp=None*, *type=None*, *user_id=None*, *app_id=None*, *cluster_id=None*)

CLASS
alias of *Basic*

INDEX = 60

NAME = 'BasicProperties'

FLAG_CONTENT_TYPE = 32768

FLAG_CONTENT_ENCODING = 16384

FLAG_HEADERS = 8192

FLAG_DELIVERY_MODE = 4096

FLAG_PRIORITY = 2048

FLAG_CORRELATION_ID = 1024

FLAG_REPLY_TO = 512

FLAG_EXPIRATION = 256

FLAG_MESSAGE_ID = 128

FLAG_TIMESTAMP = 64

FLAG_TYPE = 32

FLAG_USER_ID = 16

FLAG_APP_ID = 8

FLAG_CLUSTER_ID = 4

decode(*encoded*, *offset=0*)

encode()

pika.spec.**has_content**(*methodNumber*)

4.13 simplebuffer

Note: This class is not intended for use by those using Pika in their applications. This documentation is for those who are extending Pika or otherwise working on the driver itself.

4.13.1 SimpleBuffer

Non-module specific functions shared by modules in the pika package

pika.utils.**is_callable**(*handle*)

Returns a bool value if the handle passed in is a callable method/function

Parameters **handle** (*any*) – The object to check

Return type bool

CHAPTER 5

Authors

- Tony Garnock-Jones
- Gavin M. Roy

CHAPTER 6

Contributors

- Alexey Myasnikov
- Anton V. Yanchenko
- Ask Solem
- Asko Soukka
- Brian K. Jones
- Charles Law
- David Strauss
- Fredrik Svensson
- George y
- Hunter Morris
- Jacek ‘Forger’ Całusiński
- Jan Urbański
- Jason J. W. Williams
- Jonty Wareing
- Josh Braegger
- Josh Hansen
- Jozef Van Eenbergen
- Kamil Kisiel
- Kane
- Kyösti Herrala
- Lars van de Kerkhof
- Marek Majkowski
- Michael Kenney
- Michael Laing
- Milan Skuhra
- Njal Karevoll

- Olivier Le Thanh Duong
- Pankrat
- Pavlobaron
- Peter Magnusson
- Raphaël De Giusti
- Roey Berman
- Samuel Stauffer
- Sigurd Høgsbro

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

D

E

F

G

H

I

K

M

N

P

Q

R

S

T

U