pika

Release 0.9.13

July 21, 2015

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. Currently pika only supports Python 2.6 and 2.7. Work to support 3.3+ is underway.

If you have not developed with Pika or RabbitMQ before, the Introduction to Pika 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

2.1 Introduction to Pika

2.1.1 IO and Event Looping

As AMQP is a two-way RPC protocol where the client can send requests to the server and the server can send requests to a client, Pika implements or extends IO loops in each of its asynchronous connection adapters. These IO loops are blocking methods which loop and listen for events. Each asynchronous adapters follows the same standard for invoking the IO loop. The IO loop is created when the connection adapter is created. To start an IO loop for any given adapter, call the `connection.ioloop.start()` method.

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

Example:

```
import pika

def on_open(connection):
    # Invoked when the connection is open
    pass

# Create our connection object, passing in the on_open method
connection = pika.SelectConnection(on_open_callback=on_open)

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 *pika.credentials* module provides the mechanism by which you pass the username and password to the *ConnectionParameters* class when it is created.

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.

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
```

2.2 Core Class and Module Documentation

For the end user, Pika is organized into a small set of objects for all communication with RabbitMQ.

- A connection adapter is used to connect to RabbitMQ and manages the connection.
- Connection parameters are used to instruct the *Connection* object how to connect to RabbitMQ.
- Authentication Credentials are used to encapsulate all authentication information for the *ConnectionParameters* class.
- A *Channel* object is used to communicate with RabbitMQ via the AMQP RPC methods.
- Exceptions are raised at various points when using Pika when something goes wrong.

[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

2.2.1 Connection Adapters

Pika uses connection adapters to provide a flexible method for adapting pika's core communication to different IOLoop implementations. In addition to asynchronous adapters, there is the `BlockingConnection` adapter that provides a more idomatic procedural approach to using Pika.

Adapters

Asyncore Connection Adapter

Use Pika with the stdlib `asyncore` module.

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

The AsyncoreConnection adapter uses the stdlib asyncore module as an IOLoop for asyncronous client development.

Parameters

- **`parameters`** (*pika.connection.Parameters*) – Connection parameters
- **`on_open_callback`** (*method*) – Method to call on connection open
- **`on_open_error_callback`** (*method*) – Method to call if the connection cant be opened
- **`on_close_callback`** (*method*) – Method to call on connection close
- **`stop_ioloop_on_close`** (*bool*) – Call ioloop.stop() if disconnected

Raises RuntimeError

`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. The callback will be passed the connection, the reply_code (int) and the reply_text (str), if sent by the remote server.

Parameters **`callback_method`** (*method*) – Callback to call on close

`add_on_open_callback`(*callback_method*)

Add a callback notification when the connection has opened.

Parameters **`callback_method`** (*method*) – Callback to call when open

`add_on_open_error_callback`(*callback_method*, *remove_default=True*)

Add a callback notification when the connection can not be opened.

The callback method should accept the connection object that could not connect, and an optional error message.

Parameters

- **`callback_method`** (*method*) – Callback to call when can't connect
- **`remove_default`** (*bool*) – Remove default exception raising callback

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.

Return type *pika.channel.Channel*

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

connect()
Invoke if trying to reconnect to a RabbitMQ server. Constructing the Connection object should connect on its own.

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 blocking connection adapter module implements blocking semantics on top of Pika's core AMQP driver. While most of the asynchronous expectations are removed when using the blocking connection adapter, it attempts to remain true to the asynchronous RPC nature of the AMQP protocol, supporting server sent RPC commands.

The user facing classes in the module consist of the *BlockingConnection* and the *BlockingChannel* classes.

Be sure to check out examples in Usage Examples.

class pika.adapters.blocking_connection.**BlockingConnection**(*parameters=None*)
The BlockingConnection creates a layer on top of Pika's asynchronous core providing 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 can still 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 about communicating with the blocking_connection adapter, be sure to check out the *BlockingChannel* class which implements the *Channel* based communication for the blocking_connection adapter.

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_unused*)
This is not supported in BlockingConnection. When a connection is closed in BlockingConnection, a pika.exceptions.ConnectionClosed exception will raised instead.

Parameters **callback_method_unused** (*method*) – Unused

Raises NotImplementedError

add_on_open_callback(*callback_method_unused*)
This method is not supported in BlockingConnection.

Parameters **callback_method_unused** (*method*) – Unused

Raises NotImplementedError

add_on_open_error_callback(*callback_method_unused*, *remove_default=False*)
This method is not supported in BlockingConnection.

A pika.exceptions.AMQPConnectionError will be raised instead.

Parameters **callback_method_unused** (*method*) – Unused

Raises NotImplementedError

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

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

Return type bool

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

connect()
Invoke if trying to reconnect to a RabbitMQ server. Constructing the Connection object should connect on its own.

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.

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

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.

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.

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

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

class pika.adapters.blocking_connection.**BlockingChannel**(*connection*, *channel_number*)
The BlockingChannel implements blocking semantics for most things that one would use callback-passing-style for with the *Channel* class. In addition, the *BlockingChannel* class implements a generator that allows you to consume messages without using callbacks.

Example of creating a BlockingChannel:

```
import pika

# Create our connection object
connection = pika.BlockingConnection()

# The returned object will be a blocking channel
channel = connection.channel()
```

Parameters

- **connection** (BlockingConnection) – The connection
- **channel_number** (*int*) – The channel number for this instance

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 will receive the channel, the reply_code (int) and the reply_text (int) sent by the server describing why the channel was closed.

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(*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

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

basic_consume(*consumer_callback*, *queue=''*, *no_ack=False*, *exclusive=False*, *consumer_tag=None*, *arguments=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 or 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 or unicode*) – Specify your own consumer tag
- **arguments** (*dict*) – Custom key/value pair arguments for the consume

Return type str

basic_get(*queue=None*, *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.

Parameters

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

Return type (None, None, None)|(spec.Basic.Get, spec.Basic.Properties, str or unicode)

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. Returns a boolean value indicating the success of the operation. 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 or unicode*) – The exchange to publish to
- **routing_key** (*str or unicode*) – The routing key to bind on
- **body** (*str or unicode*) – The message body
- **properties** (*pika.spec.Properties*) – Basic.properties
- **mandatory** (*bool*) – The mandatory flag
- **immediate** (*bool*) – The immediate flag

basic_qos(*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

- **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_recover(*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 **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.

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.

cancel()
Cancel the consumption of a queue, rejecting all pending messages. This should only work with the generator based BlockingChannel.consume method. If you're looking to cancel a consumer issues with BlockingChannel.basic_consume then you should call BlockingChannel.basic_cancel.

Return int The number of messages requeued by Basic.Nack

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(*nowait=False*)
Turn on Confirm mode in the channel.

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

Parameters **nowait** (*bool*) – Do not send a reply frame (Confirm.SelectOk)

consume(*queue*, *no_ack=False*, *exclusive=False*)
Blocking consumption of a queue instead of via a callback. This method is a generator that returns messages a tuple of method, properties, and body.

Example:

for method, properties, body in channel.consume('queue'): print body channel.basic_ack(method.delivery_tag)

You should call BlockingChannel.cancel() when you escape out of the generator loop. Also note this turns on forced data events to make sure that any acked messages actually get acked.

Parameters

- **queue** (*str or unicode*) – The queue name to consume
- **no_ack** (*bool*) – Tell the broker to not expect a response
- **exclusive** (*bool*) – Don't allow other consumers on the queue

Return type tuple(spec.Basic.Deliver, spec.BasicProperties, str or unicode)

consumer_tags
Property method that returns a list of currently active consumers

Return type list

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

Parameters

- **destination** (*str or unicode*) – The destination exchange to bind
- **source** (*str or unicode*) – The source exchange to bind to
- **routing_key** (*str or 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(*exchange=None*, *exchange_type='direct'*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *nowait=False*, *arguments=None*, *type=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

- **exchange** (*str or 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
- **type** (*str*) – The deprecated exchange type parameter

exchange_delete(*exchange=None*, *if_unused=False*, *nowait=False*)
Delete the exchange.

Parameters

- **exchange** (*str or 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(*destination=None*, *source=None*, *routing_key=''*, *nowait=False*, *arguments=None*)
Unbind an exchange from another exchange.

Parameters

- **destination** (*str or unicode*) – The destination exchange to unbind
- **source** (*str or unicode*) – The source exchange to unbind from
- **routing_key** (*str or 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

force_data_events(*enable*)
Turn on and off forcing the blocking adapter to stop and look to see if there are any frames from RabbitMQ in the read buffer. By default the BlockingChannel will check for a read after every RPC command which can cause performance to degrade in scenarios where you do not care if RabbitMQ is trying to send RPC commands to your client connection.

Examples of RPC commands of this sort are:

•Heartbeats

•Connection.Close

•Channel.Close

•Basic.Return

•Basic.Ack and Basic.Nack when using delivery confirmations

Turning off forced data events can be a bad thing and prevents your client from properly communicating with RabbitMQ. Forced data events were added in 0.9.6 to enforce proper channel behavior when communicating with RabbitMQ.

Note that the BlockingConnection also has the constant WRITE_TO_READ_RATIO which forces the connection to stop and try and read after writing the number of frames specified in the constant. This is a way to force the client to received these types of frames in a very publish/write IO heavy workload.

Parameters **enable** (*bool*) – Set to False to disable

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

Parameters

- **queue** (*str or unicode*) – The queue to bind to the exchange
- **exchange** (*str or unicode*) – The source exchange to bind to
- **routing_key** (*str or 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(*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.

Leave the queue name empty for a auto-named queue in RabbitMQ

Parameters

- **queue** (*str or 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(*queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*)
Delete a queue from the broker.

Parameters

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

queue_purge(*queue=''*, *nowait=False*)
Purge all of the messages from the specified queue

Parameters

- **queue** (*str or unicode*) – The queue to purge
- **nowait** (*bool*) – Do not expect a Queue.PurgeOk response

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

Parameters

- **queue** (*str or unicode*) – The queue to unbind from the exchange
- **exchange** (*str or unicode*) – The source exchange to bind from
- **routing_key** (*str or unicode*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

start_consuming()
Starts consuming from registered callbacks.

stop_consuming(*consumer_tag=None*)
Sends off the Basic.Cancel to let RabbitMQ know to stop consuming and sets our internal state to exit out of the basic_consume.

tx_commit()
Commit a transaction.

tx_rollback()
Rollback a transaction.

tx_select()
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.

Select Connection Adapter

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*, *on_open_error_callback=None*, *on_close_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. The callback will be passed the connection, the reply_code (int) and the reply_text (str), if sent by the remote server.

Parameters **callback_method** (*method*) – Callback to call on close

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

Parameters **callback_method** (*method*) – Callback to call when open

add_on_open_error_callback(*callback_method*, *remove_default=True*)
Add a callback notification when the connection can not be opened.

The callback method should accept the connection object that could not connect, and an optional error message.

Parameters

- **callback_method** (*method*) – Callback to call when can't connect
- **remove_default** (*bool*) – Remove default exception raising callback

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.

Return type *pika.channel.Channel*

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

connect()
Invoke if trying to reconnect to a RabbitMQ server. Constructing the Connection object should connect on its own.

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

Tornado Connection Adapter

Use pika with the Tornado IOLoop

Be sure to check out the asynchronous examples including the Tornado specific consumer example.

class pika.adapters.tornado_connection.**TornadoConnection**(*parameters=None*, *on_open_callback=None*, *on_open_error_callback=None*, *on_close_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.

Parameters

- **parameters** (*pika.connection.Parameters*) – Connection parameters
- **on_open_callback** (*method*) – The method to call when the connection is open
- **on_open_error_callback** (*method*) – Method to call if the connection cant be opened
- **stop_ioloop_on_close** (*bool*) – Call ioloop.stop() if disconnected
- **custom_ioloop** – Override using the global IOLoop in Tornado

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. The callback will be passed the connection, the reply_code (int) and the reply_text (str), if sent by the remote server.

Parameters **callback_method** (*method*) – Callback to call on close

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

Parameters **callback_method** (*method*) – Callback to call when open

add_on_open_error_callback(*callback_method*, *remove_default=True*)
Add a callback notification when the connection can not be opened.

The callback method should accept the connection object that could not connect, and an optional error message.

Parameters

- **callback_method** (*method*) – Callback to call when can't connect
- **remove_default** (*bool*) – Remove default exception raising callback

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

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.

Return type *pika.channel.Channel*

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

connect()
Invoke if trying to reconnect to a RabbitMQ server. Constructing the Connection object should connect on its own.

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

Twisted Connection Adapter

2.2.2 Channel

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

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.

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 will receive the channel, the reply_code (int) and the reply_text (int) sent by the server describing why the channel was closed.

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*, *arguments=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 or 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 or unicode*) – Specify your own consumer tag
- **arguments** (*dict*) – Custom key/value pair arguments for the consume

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 or 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 or unicode*) – The exchange to publish to
- **routing_key** (*str or unicode*) – The routing key to bind on
- **body** (*str or unicode*) – The message body
- **properties** (pika.spec.BasicProperties) – 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 or unicode*) – The destination exchange to bind
- **source** (*str or unicode*) – The source exchange to bind to
- **routing_key** (*str or 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*, *type=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 or unicode sequence of these characters: letters, digits, hyphen, underscore, period, or colon.*) – The exchange name consists of a non-empty
- **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
- **type** (*str*) – The deprecated exchange type parameter

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 or 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 or unicode*) – The destination exchange to unbind
- **source** (*str or unicode*) – The source exchange to unbind from
- **routing_key** (*str or 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=None*, *nowait=False*, *arguments=None*)
Bind the queue to the specified exchange

Parameters

- **callback** (*method*) – The method to call on Queue.BindOk
- **queue** (*str or unicode*) – The queue to bind to the exchange
- **exchange** (*str or unicode*) – The source exchange to bind to
- **routing_key** (*str or 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.

Leave the queue name empty for a auto-named queue in RabbitMQ

Parameters

- **callback** (*method*) – The method to call on Queue.DeclareOk
- **queue** (*str or 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 or unicode*) – The queue to delete
- **if_unused** (*bool*) – only delete if it's unused
- **if_empty** (*bool*) – only delete if 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
- **queue** (*str or unicode*) – The queue to purge
- **nowait** (*bool*) – Do not expect a Queue.PurgeOk response

queue_unbind(*callback=None*, *queue=''*, *exchange=None*, *routing_key=None*, *arguments=None*)

Unbind a queue from an exchange.

Parameters

- **callback** (*method*) – The method to call on Queue.UnbindOk
- **queue** (*str or unicode*) – The queue to unbind from the exchange
- **exchange** (*str or unicode*) – The source exchange to bind from
- **routing_key** (*str or 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

2.2.3 Connection

The *Connection* class implements the base behavior that all connection adapters extend.

class pika.connection.**Connection**(*parameters=None*, *on_open_callback=None*, *on_open_error_callback=None*, *on_close_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.

Parameters

- **parameters** (*pika.connection.Parameters*) – Connection parameters
- **on_open_callback** (*method*) – Called when the connection is opened
- **on_open_error_callback** (*method*) – Called if the connection cant be opened
- **on_close_callback** (*method*) – Called when the connection is closed

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. The callback will be passed the connection, the reply_code (int) and the reply_text (str), if sent by the remote server.

Parameters **callback_method** (*method*) – Callback to call on close

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

Parameters **callback_method** (*method*) – Callback to call when open

add_on_open_error_callback(*callback_method*, *remove_default=True*)
Add a callback notification when the connection can not be opened.

The callback method should accept the connection object that could not connect, and an optional error message.

Parameters

- **callback_method** (*method*) – Callback to call when can't connect
- **remove_default** (*bool*) – Remove default exception raising callback

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

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.

Return type *pika.channel.Channel*

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

connect()
Invoke if trying to reconnect to a RabbitMQ server. Constructing the Connection object should connect on its own.

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(*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

2.2.4 Authentication Credentials

The credentials classes are used to encapsulate all authentication information for the *ConnectionParameters* class.

The `PlainCredentials` class returns the properly formatted username and password to the *Connection*.

To authenticate with Pika, create a `PlainCredentials` object passing in the username and password and pass it as the credentials argument value to the *ConnectionParameters* object.

If you are using *URLParameters* you do not need a credentials object, one will automatically be created for you.

If you are looking to implement SSL certificate style authentication, you would extend the `ExternalCredentials` class implementing the required behavior.

PlainCredentials

class `pika.credentials.`**PlainCredentials**(*username*, *password*, *erase_on_connect=False*)
A credentials object for the default authentication methodology with RabbitMQ.

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.

Parameters

- **username** (*str*) – The username to authenticate with
- **password** (*str*) – The password to authenticate with
- **erase_on_connect** (*bool*) – erase credentials on connect.

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)

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 or None, str or None)

2.2.5 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**
Deprecated version of UnsupportedAMQPFieldException

exception pika.exceptions.**UnsupportedAMQPFieldException**

2.2.6 Connection Parameters

To maintain flexibility in how you specify the connection information required for your applications to properly connect to RabbitMQ, pika implements two classes for encapsulating the information, *ConnectionParameters* and *URLParameters*.

ConnectionParameters

The classic object for specifying all of the connection parameters required to connect to RabbitMQ, *ConnectionParameters* provides attributes for tweaking every possible connection option.

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)
```

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.

Parameters

- **host** (*str*) – Hostname or IP Address to connect to
- **port** (*int*) – TCP port to connect to
- **virtual_host** (*str*) – RabbitMQ virtual host to use
- **credentials** (*pika.credentials.Credentials*) – auth credentials
- **channel_max** (*int*) – Maximum number of channels to allow
- **frame_max** (*int*) – The maximum byte size for an AMQP frame
- **heartbeat_interval** (*int*) – How often to send heartbeats
- **ssl** (*bool*) – Enable SSL
- **ssl_options** (*dict*) – Arguments passed to ssl.wrap_socket as
- **connection_attempts** (*int*) – Maximum number of retry attempts

- **retry_delay** (*int|float*) – Time to wait in seconds, before the next
- **socket_timeout** (*int|float*) – Use for high latency networks
- **locale** (*str*) – Set the locale value
- **backpressure_detection** (*bool*) – Toggle backpressure detection

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.

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')
```

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

Connect to RabbitMQ via an AMQP URL in the format:

```
    amqp://username:password@host:port/<virtual_host>[?query-string]
```

Ensure that the virtual host is URI encoded when specified. For example if you are using the default "/" virtual host, the value should be *%2f*.

Valid query string values are:

- **backpressure_detection:** Toggle backpressure detection, possible values are *t* or *f*
- **channel_max:** Override the default maximum channel count value
- **connection_attempts:** Specify how many times pika should try and reconnect before it gives up
- **frame_max:** Override the default maximum frame size for communication
- **heartbeat_interval:** Specify the number of seconds between heartbeat frames to ensure that the link between RabbitMQ and your application is up
- **locale:** Override the default *en_US* locale value
- **ssl:** Toggle SSL, possible values are *t*, *f*
- **ssl_options:** Arguments passed to ssl.wrap_socket()
- **retry_delay:** The number of seconds to sleep before attempting to connect on connection failure.
- **socket_timeout:** Override low level socket timeout value

Parameters **url** (*str*) – The AMQP URL to connect to

2.2.7 pika.spec

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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.**Blocked**(*reason=''*)

INDEX = 655420

NAME = 'Connection.Blocked'

synchronous

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

encode()

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.**Unblocked**

INDEX = 655421

NAME = 'Connection.Unblocked'

synchronous

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

encode()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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()

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*)

2.3 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.

2.3.1 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:

```
amqps://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:

```
amqps://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 [2].
- frame_max: Alter the default frame maximum size value by passing in a long integer value [3].
- 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

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

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

- 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'}
```

2.3.2 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.

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()
```

2.3.3 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 returns 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 *BasicProperties* 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'
```

2.3.4 Using the Blocking Connection to consume messages from RabbitMQ

The *BlockingChannel.basic_consume* method assign a callback method to be called every time that RabbitMQ delivers messages to your consuming application.

When pika calls your method, it will pass in the channel, a *pika.spec.Basic.Deliver* object with the delivery tag, 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 third argument will be a *pika.spec.BasicProperties* object and the last will be the message body.

Example of consuming messages and acknowledging them:

```
import pika


def on_message(channel, method_frame, header_frame, body):
    print method_frame.delivery_tag
    print body
    print
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)


connection = pika.BlockingConnection()
channel = connection.channel()
channel.basic_consume(on_message, 'test')
try:
    channel.start_consuming()
```

```
except KeyboardInterrupt:
    channel.stop_consuming()
connection.close()
```

2.3.5 Using the BlockingChannel.consume generator to consume messages

The *BlockingChannel.consume* method is a generator that will return a tuple of method, properties and body.

When you escape out of the loop, be sure to call consumer.cancel() to return any unprocessed messages.

Example of consuming messages and acknowledging them:

```
import pika

connection = pika.BlockingConnection()
channel = connection.channel()

# Get ten messages and break out
for method_frame, properties, body in channel.consume('test'):

    # Display the message parts
    print method_frame
    print properties
    print body

    # Acknowledge the message
    channel.basic_ack(method_frame.delivery_tag)

    # Escape out of the loop after 10 messages
    if method_frame.delivery_tag == 10:
        break

# Cancel the consumer and return any pending messages
requeued_messages = channel.cancel()
print 'Requeued %i messages' % requeued_messages

# Close the channel and the connection
channel.close()
connection.close()
```

If you have pending messages in the test queue, your output should look something like:

```
(pika)gmr-0x02:pika gmr$ python blocking_nack.py
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=1', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=2', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=3', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=4', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=5', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
```

```
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=6', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=7', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=8', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=9', 'ex
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test', 'delivery_tag=10', 'e
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
Requeued 1894 messages
```

2.3.6 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(parameters)

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()
```

2.3.7 Using Delivery Confirmations with the BlockingConnection

The following code demonstrates how to turn on delivery confirmations with the BlockingConnection and how to check for confirmation from RabbitMQ:

```
import pika

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

# Open the channel
channel = connection.channel()

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

# Turn on delivery confirmations
channel.confirm_delivery()

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

```
    print 'Message publish was confirmed'
else:
    print 'Message could not be confirmed'
```

2.3.8 Ensuring message delivery with the mandatory flag

The following example demonstrates how to check if a message is delivered by setting the mandatory flag and checking the return result when using the BlockingConnection:

```
import pika

# Open a connection to RabbitMQ on localhost using all default parameters
connection = pika.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
if channel.basic_publish(exchange='test',
                         routing_key='test',
                         body='Hello World!',
                         properties=pika.BasicProperties(content_type='text/plain',
                                                         delivery_mode=1),
                         mandatory=True):
    print 'Message was published'
else:
    print 'Message was returned'
```

2.3.9 Asynchronous consumer example

The following example implements a consumer that will respond to RPC commands sent from RabbitMQ. For example, it will reconnect if RabbitMQ closes the connection and will shutdown if RabbitMQ cancels the consumer or closes the channel. While it may look intimidating, each method is very short and represents a individual actions that a consumer can do.

consumer.py:

```
import logging
import pika

LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
              '-35s %(lineno) -5d: %(message)s')
LOGGER = logging.getLogger(__name__)


class ExampleConsumer(object):
    """This is an example consumer that will handle unexpected interactions
    with RabbitMQ such as channel and connection closures.

    If RabbitMQ closes the connection, it will reopen it. You should
    look at the output, as there are limited reasons why the connection may
    be closed, which usually are tied to permission related issues or
```

```
    socket timeouts.

    If the channel is closed, it will indicate a problem with one of the
    commands that were issued and that should surface in the output as well.

    """
    EXCHANGE = 'message'
    EXCHANGE_TYPE = 'topic'
    QUEUE = 'text'
    ROUTING_KEY = 'example.text'

    def __init__(self, amqp_url):
        """Create a new instance of the consumer class, passing in the AMQP
        URL used to connect to RabbitMQ.

        :param str amqp_url: The AMQP url to connect with

        """
        self._connection = None
        self._channel = None
        self._closing = False
        self._consumer_tag = None
        self._url = amqp_url

    def connect(self):
        """This method connects to RabbitMQ, returning the connection handle.
        When the connection is established, the on_connection_open method
        will be invoked by pika.

        :rtype: pika.SelectConnection

        """
        LOGGER.info('Connecting to %s', self._url)
        return pika.SelectConnection(pika.URLParameters(self._url),
                                     self.on_connection_open,
                                     stop_ioloop_on_close=False)

    def close_connection(self):
        """This method closes the connection to RabbitMQ."""
        LOGGER.info('Closing connection')
        self._connection.close()

    def add_on_connection_close_callback(self):
        """This method adds an on close callback that will be invoked by pika
        when RabbitMQ closes the connection to the publisher unexpectedly.

        """
        LOGGER.info('Adding connection close callback')
        self._connection.add_on_close_callback(self.on_connection_closed)

    def on_connection_closed(self, connection, reply_code, reply_text):
        """This method is invoked by pika when the connection to RabbitMQ is
        closed unexpectedly. Since it is unexpected, we will reconnect to
        RabbitMQ if it disconnects.

        :param pika.connection.Connection connection: The closed connection obj
        :param int reply_code: The server provided reply_code if given
        :param str reply_text: The server provided reply_text if given
```

```
        """
        self._channel = None
        if self._closing:
            self._connection.ioloop.stop()
        else:
            LOGGER.warning('Connection closed, reopening in 5 seconds: (%s) %s',
                           reply_code, reply_text)
            self._connection.add_timeout(5, self.reconnect)

    def on_connection_open(self, unused_connection):
        """This method is called by pika once the connection to RabbitMQ has
        been established. It passes the handle to the connection object in
        case we need it, but in this case, we'll just mark it unused.

        :type unused_connection: pika.SelectConnection

        """
        LOGGER.info('Connection opened')
        self.add_on_connection_close_callback()
        self.open_channel()

    def reconnect(self):
        """Will be invoked by the IOLoop timer if the connection is
        closed. See the on_connection_closed method.

        """
        # This is the old connection IOLoop instance, stop its ioloop
        self._connection.ioloop.stop()

        if not self._closing:

            # Create a new connection
            self._connection = self.connect()

            # There is now a new connection, needs a new ioloop to run
            self._connection.ioloop.start()

    def add_on_channel_close_callback(self):
        """This method tells pika to call the on_channel_closed method if
        RabbitMQ unexpectedly closes the channel.

        """
        LOGGER.info('Adding channel close callback')
        self._channel.add_on_close_callback(self.on_channel_closed)

    def on_channel_closed(self, channel, reply_code, reply_text):
        """Invoked by pika when RabbitMQ unexpectedly closes the channel.
        Channels are usually closed if you attempt to do something that
        violates the protocol, such as re-declare an exchange or queue with
        different parameters. In this case, we'll close the connection
        to shutdown the object.

        :param pika.channel.Channel: The closed channel
        :param int reply_code: The numeric reason the channel was closed
        :param str reply_text: The text reason the channel was closed

        """
        LOGGER.warning('Channel %i was closed: (%s) %s',
```

```
                        channel, reply_code, reply_text)
        self._connection.close()

    def on_channel_open(self, channel):
        """This method is invoked by pika when the channel has been opened.
        The channel object is passed in so we can make use of it.

        Since the channel is now open, we'll declare the exchange to use.

        :param pika.channel.Channel channel: The channel object

        """
        LOGGER.info('Channel opened')
        self._channel = channel
        self.add_on_channel_close_callback()
        self.setup_exchange(self.EXCHANGE)

    def setup_exchange(self, exchange_name):
        """Setup the exchange on RabbitMQ by invoking the Exchange.Declare RPC
        command. When it is complete, the on_exchange_declareok method will
        be invoked by pika.

        :param str|unicode exchange_name: The name of the exchange to declare

        """
        LOGGER.info('Declaring exchange %s', exchange_name)
        self._channel.exchange_declare(self.on_exchange_declareok,
                                       exchange_name,
                                       self.EXCHANGE_TYPE)

    def on_exchange_declareok(self, unused_frame):
        """Invoked by pika when RabbitMQ has finished the Exchange.Declare RPC
        command.

        :param pika.Frame.Method unused_frame: Exchange.DeclareOk response frame

        """
        LOGGER.info('Exchange declared')
        self.setup_queue(self.QUEUE)

    def setup_queue(self, queue_name):
        """Setup the queue on RabbitMQ by invoking the Queue.Declare RPC
        command. When it is complete, the on_queue_declareok method will
        be invoked by pika.

        :param str|unicode queue_name: The name of the queue to declare.

        """
        LOGGER.info('Declaring queue %s', queue_name)
        self._channel.queue_declare(self.on_queue_declareok, queue_name)

    def on_queue_declareok(self, method_frame):
        """Method invoked by pika when the Queue.Declare RPC call made in
        setup_queue has completed. In this method we will bind the queue
        and exchange together with the routing key by issuing the Queue.Bind
        RPC command. When this command is complete, the on_bindok method will
        be invoked by pika.
```

```
        :param pika.frame.Method method_frame: The Queue.DeclareOk frame

        """
        LOGGER.info('Binding %s to %s with %s',
                    self.EXCHANGE, self.QUEUE, self.ROUTING_KEY)
        self._channel.queue_bind(self.on_bindok, self.QUEUE,
                                 self.EXCHANGE, self.ROUTING_KEY)

    def add_on_cancel_callback(self):
        """Add a callback that will be invoked if RabbitMQ cancels the consumer
        for some reason. If RabbitMQ does cancel the consumer,
        on_consumer_cancelled will be invoked by pika.

        """
        LOGGER.info('Adding consumer cancellation callback')
        self._channel.add_on_cancel_callback(self.on_consumer_cancelled)

    def on_consumer_cancelled(self, method_frame):
        """Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer
        receiving messages.

        :param pika.frame.Method method_frame: The Basic.Cancel frame

        """
        LOGGER.info('Consumer was cancelled remotely, shutting down: %r',
                    method_frame)
        if self._channel:
            self._channel.close()

    def acknowledge_message(self, delivery_tag):
        """Acknowledge the message delivery from RabbitMQ by sending a
        Basic.Ack RPC method for the delivery tag.

        :param int delivery_tag: The delivery tag from the Basic.Deliver frame

        """
        LOGGER.info('Acknowledging message %s', delivery_tag)
        self._channel.basic_ack(delivery_tag)

    def on_message(self, unused_channel, basic_deliver, properties, body):
        """Invoked by pika when a message is delivered from RabbitMQ. The
        channel is passed for your convenience. The basic_deliver object that
        is passed in carries the exchange, routing key, delivery tag and
        a redelivered flag for the message. The properties passed in is an
        instance of BasicProperties with the message properties and the body
        is the message that was sent.

        :param pika.channel.Channel unused_channel: The channel object
        :param pika.Spec.Basic.Deliver: basic_deliver method
        :param pika.Spec.BasicProperties: properties
        :param str|unicode body: The message body

        """
        LOGGER.info('Received message # %s from %s: %s',
                    basic_deliver.delivery_tag, properties.app_id, body)
        self.acknowledge_message(basic_deliver.delivery_tag)

    def on_cancelok(self, unused_frame):
```

```
        """This method is invoked by pika when RabbitMQ acknowledges the
        cancellation of a consumer. At this point we will close the channel.
        This will invoke the on_channel_closed method once the channel has been
        closed, which will in-turn close the connection.

        :param pika.frame.Method unused_frame: The Basic.CancelOk frame

        """
        LOGGER.info('RabbitMQ acknowledged the cancellation of the consumer')
        self.close_channel()

    def stop_consuming(self):
        """Tell RabbitMQ that you would like to stop consuming by sending the
        Basic.Cancel RPC command.

        """
        if self._channel:
            LOGGER.info('Sending a Basic.Cancel RPC command to RabbitMQ')
            self._channel.basic_cancel(self.on_cancelok, self._consumer_tag)

    def start_consuming(self):
        """This method sets up the consumer by first calling
        add_on_cancel_callback so that the object is notified if RabbitMQ
        cancels the consumer. It then issues the Basic.Consume RPC command
        which returns the consumer tag that is used to uniquely identify the
        consumer with RabbitMQ. We keep the value to use it when we want to
        cancel consuming. The on_message method is passed in as a callback pika
        will invoke when a message is fully received.

        """
        LOGGER.info('Issuing consumer related RPC commands')
        self.add_on_cancel_callback()
        self._consumer_tag = self._channel.basic_consume(self.on_message,
                                                         self.QUEUE)

    def on_bindok(self, unused_frame):
        """Invoked by pika when the Queue.Bind method has completed. At this
        point we will start consuming messages by calling start_consuming
        which will invoke the needed RPC commands to start the process.

        :param pika.frame.Method unused_frame: The Queue.BindOk response frame

        """
        LOGGER.info('Queue bound')
        self.start_consuming()

    def close_channel(self):
        """Call to close the channel with RabbitMQ cleanly by issuing the
        Channel.Close RPC command.

        """
        LOGGER.info('Closing the channel')
        self._channel.close()

    def open_channel(self):
        """Open a new channel with RabbitMQ by issuing the Channel.Open RPC
        command. When RabbitMQ responds that the channel is open, the
        on_channel_open callback will be invoked by pika.
```

```
        """
        LOGGER.info('Creating a new channel')
        self._connection.channel(on_open_callback=self.on_channel_open)

    def run(self):
        """Run the example consumer by connecting to RabbitMQ and then
        starting the IOLoop to block and allow the SelectConnection to operate.

        """
        self._connection = self.connect()
        self._connection.ioloop.start()

    def stop(self):
        """Cleanly shutdown the connection to RabbitMQ by stopping the consumer
        with RabbitMQ. When RabbitMQ confirms the cancellation, on_cancelok
        will be invoked by pika, which will then closing the channel and
        connection. The IOLoop is started again because this method is invoked
        when CTRL-C is pressed raising a KeyboardInterrupt exception. This
        exception stops the IOLoop which needs to be running for pika to
        communicate with RabbitMQ. All of the commands issued prior to starting
        the IOLoop will be buffered but not processed.

        """
        LOGGER.info('Stopping')
        self._closing = True
        self.stop_consuming()
        self._connection.ioloop.start()
        LOGGER.info('Stopped')


def main():
    logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
    example = ExampleConsumer('amqp://guest:guest@localhost:5672/%2F')
    try:
        example.run()
    except KeyboardInterrupt:
        example.stop()


if __name__ == '__main__':
    main()
```

2.3.10 Asynchronous publisher example

The following example implements a publisher that will respond to RPC commands sent from RabbitMQ and uses delivery confirmations. It will reconnect if RabbitMQ closes the connection and will shutdown if RabbitMQ closes the channel. While it may look intimidating, each method is very short and represents a individual actions that a publisher can do.

publisher.py:

```
# -*- coding: utf-8 -*-
import logging
import pika
import json
```

```
LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
              '-35s %(lineno) -5d: %(message)s')
LOGGER = logging.getLogger(__name__)


class ExamplePublisher(object):
    """This is an example publisher that will handle unexpected interactions
    with RabbitMQ such as channel and connection closures.

    If RabbitMQ closes the connection, it will reopen it. You should
    look at the output, as there are limited reasons why the connection may
    be closed, which usually are tied to permission related issues or
    socket timeouts.

    It uses delivery confirmations and illustrates one way to keep track of
    messages that have been sent and if they've been confirmed by RabbitMQ.

    """
    EXCHANGE = 'message'
    EXCHANGE_TYPE = 'topic'
    PUBLISH_INTERVAL = 1
    QUEUE = 'text'
    ROUTING_KEY = 'example.text'

    def __init__(self, amqp_url):
        """Setup the example publisher object, passing in the URL we will use
        to connect to RabbitMQ.

        :param str amqp_url: The URL for connecting to RabbitMQ

        """
        self._connection = None
        self._channel = None
        self._deliveries = []
        self._acked = 0
        self._nacked = 0
        self._message_number = 0
        self._stopping = False
        self._url = amqp_url
        self._closing = False

    def connect(self):
        """This method connects to RabbitMQ, returning the connection handle.
        When the connection is established, the on_connection_open method
        will be invoked by pika. If you want the reconnection to work, make
        sure you set stop_ioloop_on_close to False, which is not the default
        behavior of this adapter.

        :rtype: pika.SelectConnection

        """
        LOGGER.info('Connecting to %s', self._url)
        return pika.SelectConnection(pika.URLParameters(self._url),
                                     self.on_connection_open,
                                     stop_ioloop_on_close=False)

    def close_connection(self):
        """This method closes the connection to RabbitMQ."""
```

```
        LOGGER.info('Closing connection')
        self._closing = True
        self._connection.close()

    def add_on_connection_close_callback(self):
        """This method adds an on close callback that will be invoked by pika
        when RabbitMQ closes the connection to the publisher unexpectedly.

        """
        LOGGER.info('Adding connection close callback')
        self._connection.add_on_close_callback(self.on_connection_closed)

    def on_connection_closed(self, connection, reply_code, reply_text):
        """This method is invoked by pika when the connection to RabbitMQ is
        closed unexpectedly. Since it is unexpected, we will reconnect to
        RabbitMQ if it disconnects.

        :param pika.connection.Connection connection: The closed connection obj
        :param int reply_code: The server provided reply_code if given
        :param str reply_text: The server provided reply_text if given

        """
        self._channel = None
        if self._closing:
            self._connection.ioloop.stop()
        else:
            LOGGER.warning('Connection closed, reopening in 5 seconds: (%s) %s',
                           reply_code, reply_text)
            self._connection.add_timeout(5, self.reconnect)

    def on_connection_open(self, unused_connection):
        """This method is called by pika once the connection to RabbitMQ has
        been established. It passes the handle to the connection object in
        case we need it, but in this case, we'll just mark it unused.

        :type unused_connection: pika.SelectConnection

        """
        LOGGER.info('Connection opened')
        self.add_on_connection_close_callback()
        self.open_channel()

    def reconnect(self):
        """Will be invoked by the IOLoop timer if the connection is
        closed. See the on_connection_closed method.

        """
        # This is the old connection IOLoop instance, stop its ioloop
        self._connection.ioloop.stop()

        # Create a new connection
        self._connection = self.connect()

        # There is now a new connection, needs a new ioloop to run
        self._connection.ioloop.start()

    def add_on_channel_close_callback(self):
        """This method tells pika to call the on_channel_closed method if
```

```
        RabbitMQ unexpectedly closes the channel.

        """
        LOGGER.info('Adding channel close callback')
        self._channel.add_on_close_callback(self.on_channel_closed)

    def on_channel_closed(self, channel, reply_code, reply_text):
        """Invoked by pika when RabbitMQ unexpectedly closes the channel.
        Channels are usually closed if you attempt to do something that
        violates the protocol, such as re-declare an exchange or queue with
        different parameters. In this case, we'll close the connection
        to shutdown the object.

        :param pika.channel.Channel: The closed channel
        :param int reply_code: The numeric reason the channel was closed
        :param str reply_text: The text reason the channel was closed

        """
        LOGGER.warning('Channel was closed: (%s) %s', reply_code, reply_text)
        if not self._closing:
            self._connection.close()

    def on_channel_open(self, channel):
        """This method is invoked by pika when the channel has been opened.
        The channel object is passed in so we can make use of it.

        Since the channel is now open, we'll declare the exchange to use.

        :param pika.channel.Channel channel: The channel object

        """
        LOGGER.info('Channel opened')
        self._channel = channel
        self.add_on_channel_close_callback()
        self.setup_exchange(self.EXCHANGE)

    def setup_exchange(self, exchange_name):
        """Setup the exchange on RabbitMQ by invoking the Exchange.Declare RPC
        command. When it is complete, the on_exchange_declareok method will
        be invoked by pika.

        :param str|unicode exchange_name: The name of the exchange to declare

        """
        LOGGER.info('Declaring exchange %s', exchange_name)
        self._channel.exchange_declare(self.on_exchange_declareok,
                                       exchange_name,
                                       self.EXCHANGE_TYPE)

    def on_exchange_declareok(self, unused_frame):
        """Invoked by pika when RabbitMQ has finished the Exchange.Declare RPC
        command.

        :param pika.Frame.Method unused_frame: Exchange.DeclareOk response frame

        """
        LOGGER.info('Exchange declared')
        self.setup_queue(self.QUEUE)
```

```
    def setup_queue(self, queue_name):
        """Setup the queue on RabbitMQ by invoking the Queue.Declare RPC
        command. When it is complete, the on_queue_declareok method will
        be invoked by pika.

        :param str|unicode queue_name: The name of the queue to declare.

        """
        LOGGER.info('Declaring queue %s', queue_name)
        self._channel.queue_declare(self.on_queue_declareok, queue_name)

    def on_queue_declareok(self, method_frame):
        """Method invoked by pika when the Queue.Declare RPC call made in
        setup_queue has completed. In this method we will bind the queue
        and exchange together with the routing key by issuing the Queue.Bind
        RPC command. When this command is complete, the on_bindok method will
        be invoked by pika.

        :param pika.frame.Method method_frame: The Queue.DeclareOk frame

        """
        LOGGER.info('Binding %s to %s with %s',
                    self.EXCHANGE, self.QUEUE, self.ROUTING_KEY)
        self._channel.queue_bind(self.on_bindok, self.QUEUE,
                                 self.EXCHANGE, self.ROUTING_KEY)

    def on_delivery_confirmation(self, method_frame):
        """Invoked by pika when RabbitMQ responds to a Basic.Publish RPC
        command, passing in either a Basic.Ack or Basic.Nack frame with
        the delivery tag of the message that was published. The delivery tag
        is an integer counter indicating the message number that was sent
        on the channel via Basic.Publish. Here we're just doing house keeping
        to keep track of stats and remove message numbers that we expect
        a delivery confirmation of from the list used to keep track of messages
        that are pending confirmation.

        :param pika.frame.Method method_frame: Basic.Ack or Basic.Nack frame

        """
        confirmation_type = method_frame.method.NAME.split('.')[1].lower()
        LOGGER.info('Received %s for delivery tag: %i',
                    confirmation_type,
                    method_frame.method.delivery_tag)
        if confirmation_type == 'ack':
            self._acked += 1
        elif confirmation_type == 'nack':
            self._nacked += 1
        self._deliveries.remove(method_frame.method.delivery_tag)
        LOGGER.info('Published %i messages, %i have yet to be confirmed, '
                    '%i were acked and %i were nacked',
                    self._message_number, len(self._deliveries),
                    self._acked, self._nacked)

    def enable_delivery_confirmations(self):
        """Send the Confirm.Select RPC method to RabbitMQ to enable delivery
        confirmations on the channel. The only way to turn this off is to close
        the channel and create a new one.
```

```
        When the message is confirmed from RabbitMQ, the
        on_delivery_confirmation method will be invoked passing in a Basic.Ack
        or Basic.Nack method from RabbitMQ that will indicate which messages it
        is confirming or rejecting.

        """
        LOGGER.info('Issuing Confirm.Select RPC command')
        self._channel.confirm_delivery(self.on_delivery_confirmation)

    def publish_message(self):
        """If the class is not stopping, publish a message to RabbitMQ,
        appending a list of deliveries with the message number that was sent.
        This list will be used to check for delivery confirmations in the
        on_delivery_confirmations method.

        Once the message has been sent, schedule another message to be sent.
        The main reason I put scheduling in was just so you can get a good idea
        of how the process is flowing by slowing down and speeding up the
        delivery intervals by changing the PUBLISH_INTERVAL constant in the
        class.

        """
        if self._stopping:
            return

        message = {u'': u' ',
                   u'': u'',
                   u'': u''}
        properties = pika.BasicProperties(app_id='example-publisher',
                                          content_type='application/json',
                                          headers=message)

        self._channel.basic_publish(self.EXCHANGE, self.ROUTING_KEY,
                                    json.dumps(message, ensure_ascii=False),
                                    properties)
        self._message_number += 1
        self._deliveries.append(self._message_number)
        LOGGER.info('Published message # %i', self._message_number)
        self.schedule_next_message()

    def schedule_next_message(self):
        """If we are not closing our connection to RabbitMQ, schedule another
        message to be delivered in PUBLISH_INTERVAL seconds.

        """
        if self._stopping:
            return
        LOGGER.info('Scheduling next message for %0.1f seconds',
                    self.PUBLISH_INTERVAL)
        self._connection.add_timeout(self.PUBLISH_INTERVAL,
                                     self.publish_message)

    def start_publishing(self):
        """This method will enable delivery confirmations and schedule the
        first message to be sent to RabbitMQ

        """
        LOGGER.info('Issuing consumer related RPC commands')
```

```
        self.enable_delivery_confirmations()
        self.schedule_next_message()

    def on_bindok(self, unused_frame):
        """This method is invoked by pika when it receives the Queue.BindOk
        response from RabbitMQ. Since we know we're now setup and bound, it's
        time to start publishing."""
        LOGGER.info('Queue bound')
        self.start_publishing()

    def close_channel(self):
        """Invoke this command to close the channel with RabbitMQ by sending
        the Channel.Close RPC command.

        """
        LOGGER.info('Closing the channel')
        if self._channel:
            self._channel.close()

    def open_channel(self):
        """This method will open a new channel with RabbitMQ by issuing the
        Channel.Open RPC command. When RabbitMQ confirms the channel is open
        by sending the Channel.OpenOK RPC reply, the on_channel_open method
        will be invoked.

        """
        LOGGER.info('Creating a new channel')
        self._connection.channel(on_open_callback=self.on_channel_open)

    def run(self):
        """Run the example code by connecting and then starting the IOLoop.

        """
        self._connection = self.connect()
        self._connection.ioloop.start()

    def stop(self):
        """Stop the example by closing the channel and connection. We
        set a flag here so that we stop scheduling new messages to be
        published. The IOLoop is started because this method is
        invoked by the Try/Catch below when KeyboardInterrupt is caught.
        Starting the IOLoop again will allow the publisher to cleanly
        disconnect from RabbitMQ.

        """
        LOGGER.info('Stopping')
        self._stopping = True
        self.close_channel()
        self.close_connection()
        self._connection.ioloop.start()
        LOGGER.info('Stopped')

def main():
    logging.basicConfig(level=logging.DEBUG, format=LOG_FORMAT)

    # Connect to localhost:5672 as guest with the password guest and virtual host "/" (%2F)
    example = ExamplePublisher('amqp://guest:guest@localhost:5672/%2F?connection_attempts=3&heartbeat
    try:
```

```
        example.run()
    except KeyboardInterrupt:
        example.stop()

if __name__ == '__main__':
    main()
```

2.3.11 Twisted Consumer Example

Example of writing a consumer using the `Twisted connection adapter`:

```
# -*- coding:utf-8 -*-

import pika
from pika import exceptions
from pika.adapters import twisted_connection
from twisted.internet import defer, reactor, protocol,task


@defer.inlineCallbacks
def run(connection):

    channel = yield connection.channel()

    exchange = yield channel.exchange_declare(exchange='topic_link',type='topic')

    queue = yield channel.queue_declare(queue='hello', auto_delete=False, exclusive=False)

    yield channel.queue_bind(exchange='topic_link',queue='hello',routing_key='hello.world')

    yield channel.basic_qos(prefetch_count=1)

    queue_object, consumer_tag = yield channel.basic_consume(queue='hello',no_ack=False)

    l = task.LoopingCall(read, queue_object)

    l.start(0.01)


@defer.inlineCallbacks
def read(queue_object):

    ch,method,properties,body = yield queue_object.get()

    if body:
        print body

    yield ch.basic_ack(delivery_tag=method.delivery_tag)


parameters = pika.ConnectionParameters()
cc = protocol.ClientCreator(reactor, twisted_connection.TwistedProtocolConnection, parameters)
d = cc.connectTCP('hostname', 5672)
d.addCallback(lambda protocol: protocol.ready)
d.addCallback(run)
reactor.run()
```

2.3.12 Tornado Consumer

The following example implements a consumer using the *Tornado adapter* for the Tornado framework that will respond to RPC commands sent from RabbitMQ. For example, it will reconnect if RabbitMQ closes the connection and will shutdown if RabbitMQ cancels the consumer or closes the channel. While it may look intimidating, each method is very short and represents a individual actions that a consumer can do.

consumer.py:

```
from pika import adapters
import pika
import logging

LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
              '-35s %(lineno) -5d: %(message)s')
LOGGER = logging.getLogger(__name__)


class ExampleConsumer(object):
    """This is an example consumer that will handle unexpected interactions
    with RabbitMQ such as channel and connection closures.

    If RabbitMQ closes the connection, it will reopen it. You should
    look at the output, as there are limited reasons why the connection may
    be closed, which usually are tied to permission related issues or
    socket timeouts.

    If the channel is closed, it will indicate a problem with one of the
    commands that were issued and that should surface in the output as well.

    """
    EXCHANGE = 'message'
    EXCHANGE_TYPE = 'topic'
    QUEUE = 'text'
    ROUTING_KEY = 'example.text'

    def __init__(self, amqp_url):
        """Create a new instance of the consumer class, passing in the AMQP
        URL used to connect to RabbitMQ.

        :param str amqp_url: The AMQP url to connect with

        """
        self._connection = None
        self._channel = None
        self._closing = False
        self._consumer_tag = None
        self._url = amqp_url

    def connect(self):
        """This method connects to RabbitMQ, returning the connection handle.
        When the connection is established, the on_connection_open method
        will be invoked by pika.

        :rtype: pika.SelectConnection

        """
        LOGGER.info('Connecting to %s', self._url)
        return adapters.TornadoConnection(pika.URLParameters(self._url),
```

```
                                        self.on_connection_open)

    def close_connection(self):
        """This method closes the connection to RabbitMQ."""
        LOGGER.info('Closing connection')
        self._connection.close()

    def add_on_connection_close_callback(self):
        """This method adds an on close callback that will be invoked by pika
        when RabbitMQ closes the connection to the publisher unexpectedly.

        """
        LOGGER.info('Adding connection close callback')
        self._connection.add_on_close_callback(self.on_connection_closed)

    def on_connection_closed(self, connection, reply_code, reply_text):
        """This method is invoked by pika when the connection to RabbitMQ is
        closed unexpectedly. Since it is unexpected, we will reconnect to
        RabbitMQ if it disconnects.

        :param pika.connection.Connection connection: The closed connection obj
        :param int reply_code: The server provided reply_code if given
        :param str reply_text: The server provided reply_text if given

        """
        self._channel = None
        if self._closing:
            self._connection.ioloop.stop()
        else:
            LOGGER.warning('Connection closed, reopening in 5 seconds: (%s) %s',
                           reply_code, reply_text)
            self._connection.add_timeout(5, self.reconnect)

    def on_connection_open(self, unused_connection):
        """This method is called by pika once the connection to RabbitMQ has
        been established. It passes the handle to the connection object in
        case we need it, but in this case, we'll just mark it unused.

        :type unused_connection: pika.SelectConnection

        """
        LOGGER.info('Connection opened')
        self.add_on_connection_close_callback()
        self.open_channel()

    def reconnect(self):
        """Will be invoked by the IOLoop timer if the connection is
        closed. See the on_connection_closed method.

        """
        if not self._closing:

            # Create a new connection
            self._connection = self.connect()

    def add_on_channel_close_callback(self):
        """This method tells pika to call the on_channel_closed method if
        RabbitMQ unexpectedly closes the channel.
```

```
        """
        LOGGER.info('Adding channel close callback')
        self._channel.add_on_close_callback(self.on_channel_closed)

    def on_channel_closed(self, channel, reply_code, reply_text):
        """Invoked by pika when RabbitMQ unexpectedly closes the channel.
        Channels are usually closed if you attempt to do something that
        violates the protocol, such as re-declare an exchange or queue with
        different parameters. In this case, we'll close the connection
        to shutdown the object.

        :param pika.channel.Channel: The closed channel
        :param int reply_code: The numeric reason the channel was closed
        :param str reply_text: The text reason the channel was closed

        """
        LOGGER.warning('Channel %i was closed: (%s) %s',
                       channel, reply_code, reply_text)
        self._connection.close()

    def on_channel_open(self, channel):
        """This method is invoked by pika when the channel has been opened.
        The channel object is passed in so we can make use of it.

        Since the channel is now open, we'll declare the exchange to use.

        :param pika.channel.Channel channel: The channel object

        """
        LOGGER.info('Channel opened')
        self._channel = channel
        self.add_on_channel_close_callback()
        self.setup_exchange(self.EXCHANGE)

    def setup_exchange(self, exchange_name):
        """Setup the exchange on RabbitMQ by invoking the Exchange.Declare RPC
        command. When it is complete, the on_exchange_declareok method will
        be invoked by pika.

        :param str|unicode exchange_name: The name of the exchange to declare

        """
        LOGGER.info('Declaring exchange %s', exchange_name)
        self._channel.exchange_declare(self.on_exchange_declareok,
                                       exchange_name,
                                       self.EXCHANGE_TYPE)

    def on_exchange_declareok(self, unused_frame):
        """Invoked by pika when RabbitMQ has finished the Exchange.Declare RPC
        command.

        :param pika.Frame.Method unused_frame: Exchange.DeclareOk response frame

        """
        LOGGER.info('Exchange declared')
        self.setup_queue(self.QUEUE)

    def setup_queue(self, queue_name):
```

```
        """Setup the queue on RabbitMQ by invoking the Queue.Declare RPC
        command. When it is complete, the on_queue_declareok method will
        be invoked by pika.

        :param str|unicode queue_name: The name of the queue to declare.

        """
        LOGGER.info('Declaring queue %s', queue_name)
        self._channel.queue_declare(self.on_queue_declareok, queue_name)

    def on_queue_declareok(self, method_frame):
        """Method invoked by pika when the Queue.Declare RPC call made in
        setup_queue has completed. In this method we will bind the queue
        and exchange together with the routing key by issuing the Queue.Bind
        RPC command. When this command is complete, the on_bindok method will
        be invoked by pika.

        :param pika.frame.Method method_frame: The Queue.DeclareOk frame

        """
        LOGGER.info('Binding %s to %s with %s',
                    self.EXCHANGE, self.QUEUE, self.ROUTING_KEY)
        self._channel.queue_bind(self.on_bindok, self.QUEUE,
                                 self.EXCHANGE, self.ROUTING_KEY)

    def add_on_cancel_callback(self):
        """Add a callback that will be invoked if RabbitMQ cancels the consumer
        for some reason. If RabbitMQ does cancel the consumer,
        on_consumer_cancelled will be invoked by pika.

        """
        LOGGER.info('Adding consumer cancellation callback')
        self._channel.add_on_cancel_callback(self.on_consumer_cancelled)

    def on_consumer_cancelled(self, method_frame):
        """Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer
        receiving messages.

        :param pika.frame.Method method_frame: The Basic.Cancel frame

        """
        LOGGER.info('Consumer was cancelled remotely, shutting down: %r',
                    method_frame)
        if self._channel:
            self._channel.close()

    def acknowledge_message(self, delivery_tag):
        """Acknowledge the message delivery from RabbitMQ by sending a
        Basic.Ack RPC method for the delivery tag.

        :param int delivery_tag: The delivery tag from the Basic.Deliver frame

        """
        LOGGER.info('Acknowledging message %s', delivery_tag)
        self._channel.basic_ack(delivery_tag)

    def on_message(self, unused_channel, basic_deliver, properties, body):
        """Invoked by pika when a message is delivered from RabbitMQ. The
```

```
        channel is passed for your convenience. The basic_deliver object that
        is passed in carries the exchange, routing key, delivery tag and
        a redelivered flag for the message. The properties passed in is an
        instance of BasicProperties with the message properties and the body
        is the message that was sent.

        :param pika.channel.Channel unused_channel: The channel object
        :param pika.Spec.Basic.Deliver: basic_deliver method
        :param pika.Spec.BasicProperties: properties
        :param str|unicode body: The message body

        """
        LOGGER.info('Received message # %s from %s: %s',
                    basic_deliver.delivery_tag, properties.app_id, body)
        self.acknowledge_message(basic_deliver.delivery_tag)

    def on_cancelok(self, unused_frame):
        """This method is invoked by pika when RabbitMQ acknowledges the
        cancellation of a consumer. At this point we will close the channel.
        This will invoke the on_channel_closed method once the channel has been
        closed, which will in-turn close the connection.

        :param pika.frame.Method unused_frame: The Basic.CancelOk frame

        """
        LOGGER.info('RabbitMQ acknowledged the cancellation of the consumer')
        self.close_channel()

    def stop_consuming(self):
        """Tell RabbitMQ that you would like to stop consuming by sending the
        Basic.Cancel RPC command.

        """
        if self._channel:
            LOGGER.info('Sending a Basic.Cancel RPC command to RabbitMQ')
            self._channel.basic_cancel(self.on_cancelok, self._consumer_tag)

    def start_consuming(self):
        """This method sets up the consumer by first calling
        add_on_cancel_callback so that the object is notified if RabbitMQ
        cancels the consumer. It then issues the Basic.Consume RPC command
        which returns the consumer tag that is used to uniquely identify the
        consumer with RabbitMQ. We keep the value to use it when we want to
        cancel consuming. The on_message method is passed in as a callback pika
        will invoke when a message is fully received.

        """
        LOGGER.info('Issuing consumer related RPC commands')
        self.add_on_cancel_callback()
        self._consumer_tag = self._channel.basic_consume(self.on_message,
                                                         self.QUEUE)

    def on_bindok(self, unused_frame):
        """Invoked by pika when the Queue.Bind method has completed. At this
        point we will start consuming messages by calling start_consuming
        which will invoke the needed RPC commands to start the process.

        :param pika.frame.Method unused_frame: The Queue.BindOk response frame
```

```
        """
        LOGGER.info('Queue bound')
        self.start_consuming()

    def close_channel(self):
        """Call to close the channel with RabbitMQ cleanly by issuing the
        Channel.Close RPC command.

        """
        LOGGER.info('Closing the channel')
        self._channel.close()

    def open_channel(self):
        """Open a new channel with RabbitMQ by issuing the Channel.Open RPC
        command. When RabbitMQ responds that the channel is open, the
        on_channel_open callback will be invoked by pika.

        """
        LOGGER.info('Creating a new channel')
        self._connection.channel(on_open_callback=self.on_channel_open)

    def run(self):
        """Run the example consumer by connecting to RabbitMQ and then
        starting the IOLoop to block and allow the SelectConnection to operate.

        """
        self._connection = self.connect()
        self._connection.ioloop.start()

    def stop(self):
        """Cleanly shutdown the connection to RabbitMQ by stopping the consumer
        with RabbitMQ. When RabbitMQ confirms the cancellation, on_cancelok
        will be invoked by pika, which will then closing the channel and
        connection. The IOLoop is started again because this method is invoked
        when CTRL-C is pressed raising a KeyboardInterrupt exception. This
        exception stops the IOLoop which needs to be running for pika to
        communicate with RabbitMQ. All of the commands issued prior to starting
        the IOLoop will be buffered but not processed.

        """
        LOGGER.info('Stopping')
        self._closing = True
        self.stop_consuming()
        self._connection.ioloop.start()
        LOGGER.info('Stopped')


def main():
    logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
    example = ExampleConsumer('amqp://guest:guest@localhost:5672/%2F')
    try:
        example.run()
    except KeyboardInterrupt:
        example.stop()


if __name__ == '__main__':
    main()
```

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

- Does Pika 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.

2.5 Contributors

The following people have directly contributes code by way of new features and/or bug fixes to Pika:

- Adam Flynn
- Ales Teska
- Alexey Myasnikov
- Anton V. Yanchenko
- Ask Solem
- Asko Soukka
- Brian K. Jones
- Cenk Altı
- Charles Law
- David Strauss
- Erik Andersson
- Fredrik Svensson

- Garth Williamson
- 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
- Mark Unsworth
- Michael Kenney
- Michael Laing
- Milan Skuhra
- Njal Karevoll
- nonleaf
- Olivier Le Thanh Duong
- Pankrat
- Pavlobaron
- Peter Magnusson
- Ralf Nyrén
- Raphaël De Giusti
- Rikard Hultén
- Roey Berman
- Samuel Stauffer
- Sigurd Høgsbro
- Tristan Penman

2.6 Version History

2.6.1 0.9.13 - 2013-05-15

Major Changes

- IPv6 Support with thanks to Alessandro Tagliapietra for initial prototype
- Officially remove support for <= Python 2.5 even though it was broken already
- Drop pika.simplebuffer.SimpleBuffer in favor of the Python stdlib collections.deque object
- New default object for receiving content is a "bytes" object which is a str wrapper in Python 2, but paves way for Python 3 support
- New "Raw" mode for frame decoding content frames (#334) addresses issues #331, #229 added by Garth Williamson
- Connection and Disconnection logic refactored, allowing for cleaner separation of protocol logic and socket handling logic as well as connection state management
- New "on_open_error_callback" argument in creating connection objects and new Connection.add_on_open_error_callback method
- New Connection.connect method to cleanly allow for reconnection code
- Support for all AMQP field types, using protocol specified signed/unsigned unpacking

Backwards Incompatible Changes

- Method signature for creating connection objects has new argument "on_open_error_callback" which is positionally before "on_close_callback"
- Internal callback variable names in connection.Connection have been renamed and constants used. If you relied on any of these callbacks outside of their internal use, make sure to check out the new constants.
- Connection._connect method, which was an internal only method is now deprecated and will raise a DeprecationWarning. If you relied on this method, your code needs to change.
- pika.simplebuffer has been removed

Bugfixes

- BlockingConnection consumer generator does not free buffer when exited (#328)
- Unicode body payloads in the blocking adapter raises exception (#333)
- Support "b" short-short-int AMQP data type (#318)
- Docstring type fix in adapters/select_connection (#316) fix by Rikard Hultén
- IPv6 not supported (#309)
- Stop the HeartbeatChecker when connection is closed (#307)
- Unittest fix for SelectConnection (#336) fix by Erik Andersson
- Handle condition where no connection or socket exists but SelectConnection needs a timeout for retrying a connection (#322)
- TwistedAdapter lagging behind BaseConnection changes (#321) fix by Jan Urbański

Other

- Refactored documentation

- Added Twisted Adapter example (#314) by nolinksoft

2.6.2 0.9.12 - 2013-03-18

Bugfixes

- New timeout id hashing was not unique

2.6.3 0.9.11 - 2013-03-17

Bugfixes

- Address inconsistent channel close callback documentation and add the signature change to the TwistedChannel class (#305)
- Address a missed timeout related internal data structure name change introduced in the SelectConnection 0.9.10 release. Update all connection adapters to use same signature and docstring (#306).

2.6.4 0.9.10 - 2013-03-16

Bugfixes

- Fix timeout in twisted adapter (Submitted by cellscape)
- Fix blocking_connection poll timer resolution to milliseconds (Submitted by cellscape)
- Fix channel._on_close() without a method frame (Submitted by Richard Boulton)
- Addressed exception on close (Issue #279 - fix by patcpsc)
- 'messages' not initialized in BlockingConnection.cancel() (Issue #289 - fix by Mik Kocikowski)
- Make queue_unbind behave like queue_bind (Issue #277)
- Address closing behavioral issues for connections and channels (Issue #275)
- Pass a Method frame to Channel._on_close in Connection._on_disconnect (Submitted by Jan Urbański)
- Fix channel closed callback signature in the Twisted adapter (Submitted by Jan Urbański)
- Don't stop the IOLoop on connection close for in the Twisted adapter (Submitted by Jan Urbański)
- Update the asynchronous examples to fix reconnecting and have it work
- Warn if the socket was closed such as if RabbitMQ dies without a Close frame
- Fix URLParameters ssl_options (Issue #296)
- Add state to BlockingConnection addressing (Issue #301)
- Encode unicode body content prior to publishing (Issue #282)
- Fix an issue with unicode keys in BasicProperties headers key (Issue #280)
- Change how timeout ids are generated (Issue #254)
- Address post close state issues in Channel (Issue #302)

** Behavior changes **

- Change core connection communication behavior to prefer outbound writes over reads, addressing a recursion issue

- Update connection on close callbacks, changing callback method signature
- Update channel on close callbacks, changing callback method signature
- Give more info in the ChannelClosed exception
- Change the constructor signature for BlockingConnection, block open/close callbacks
- Disable the use of add_on_open_callback/add_on_close_callback methods in BlockingConnection

2.6.5 0.9.9 - 2013-01-29

Bugfixes

- Only remove the tornado_connection.TornadoConnection file descriptor from the IOLoop if it's still open (Issue #221)
- Allow messages with no body (Issue #227)
- Allow for empty routing keys (Issue #224)
- Don't raise an exception when trying to send a frame to a closed connection (Issue #229)
- Only send a Connection.CloseOk if the connection is still open. (Issue #236 - Fix by noleaf)
- Fix timeout threshold in blocking connection - (Issue #232 - Fix by Adam Flynn)
- Fix closing connection while a channel is still open (Issue #230 - Fix by Adam Flynn)
- Fixed misleading warning and exception messages in BaseConnection (Issue #237 - Fix by Tristan Penman)
- Pluralised and altered the wording of the AMQPConnectionError exception (Issue #237 - Fix by Tristan Penman)
- Fixed _adapter_disconnect in TornadoConnection class (Issue #237 - Fix by Tristan Penman)
- Fixing hang when closing connection without any channel in BlockingConnection (Issue #244 - Fix by Ales Teska)
- Remove the process_timeouts() call in SelectConnection (Issue #239)
- Change the string validation to basestring for host connection parameters (Issue #231)
- Add a poller to the BlockingConnection to address latency issues introduced in Pika 0.9.8 (Issue #242)
- reply_code and reply_text is not set in ChannelException (Issue #250)
- Add the missing constraint parameter for Channel._on_return callback processing (Issue #257 - Fix by patcpsc)
- Channel callbacks not being removed from callback manager when channel is closed or deleted (Issue #261)

2.6.6 0.9.8 - 2012-11-18

Bugfixes

- Channel.queue_declare/BlockingChannel.queue_declare not setting up callbacks property for empty queue name (Issue #218)
- Channel.queue_bind/BlockingChannel.queue_bind not allowing empty routing key
- Connection._on_connection_closed calling wrong method in Channel (Issue #219)
- Fix tx_commit and tx_rollback bugs in BlockingChannel (Issue #217)

2.6.7 0.9.7 - 2012-11-11

New features

- generator based consumer in BlockingChannel (See Using the BlockingChannel.consume generator to consume messages for example)

Changes

- BlockingChannel._send_method will only wait if explicitly told to

Bugfixes

- Added the exchange "type" parameter back but issue a DeprecationWarning
- Dont require a queue name in Channel.queue_declare()
- Fixed KeyError when processing timeouts (Issue # 215 - Fix by Raphael De Giusti)
- Don't try and close channels when the connection is closed (Issue #216 - Fix by Charles Law)
- Dont raise UnexpectedFrame exceptions, log them instead
- Handle multiple synchronous RPC calls made without waiting for the call result (Issues #192, #204, #211)
- Typo in docs (Issue #207 Fix by Luca Wehrstedt)
- Only sleep on connection failure when retry attempts are > 0 (Issue #200)
- Bypass _rpc method and just send frames for Basic.Ack, Basic.Nack, Basic.Reject (Issue #205)

2.6.8 0.9.6 - 2012-10-29

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

2.6.9 0.9.5 - 2011-03-29

Changelog

- Scope changes with adapter IOLoops and CallbackManager allowing for cleaner, multi-threaded operation
- Add support for Confirm.Select with channel.Channel.confirm_delivery()
- Add examples of delivery confirmation to examples (demo_send_confirmed.py)
- Update uses of log.warn with warning.warn for TCP Back-pressure alerting
- License boilerplate updated to simplify license text in source files
- Increment the timeout in select_connection.SelectPoller reducing CPU utilization
- Bug fix in Heartbeat frame delivery addressing issue #35
- Remove abuse of pika.log.method_call through a majority of the code
- Rename of key modules: table to data, frames to frame
- Cleanup of frame module and related classes
- Restructure of tests and test runner
- Update functional tests to respect RABBITMQ_HOST, RABBITMQ_PORT environment variables
- Bug fixes to reconnection_strategies module
- Fix the scale of timeout for PollPoller to be specified in milliseconds
- Remove mutable default arguments in RPC calls
- Add data type validation to RPC calls
- Move optional credentials erasing out of connection.Connection into credentials module
- Add support to allow for additional external credential types
- Add a NullHandler to prevent the 'No handlers could be found for logger "pika"' error message when not using pika.log in a client app at all.
- Clean up all examples to make them easier to read and use
- Move documentation into its own repository https://github.com/pika/documentation
- channel.py
  - Move channel.MAX_CHANNELS constant from connection.CHANNEL_MAX
  - Add default value of None to ChannelTransport.rpc
  - Validate callback and acceptable replies parameters in ChannelTransport.RPC
  - Remove unused connection attribute from Channel
- connection.py
  - Remove unused import of struct
  - Remove direct import of pika.credentials.PlainCredentials - Change to import pika.credentials
  - Move CHANNEL_MAX to channel.MAX_CHANNELS
  - Change ConnectionParameters initialization parameter heartbeat to boolean
  - Validate all inbound parameter types in ConnectionParameters

  - Remove the Connection._erase_credentials stub method in favor of letting the Credentials object deal with that itself.
  - Warn if the credentials object intends on erasing the credentials and a reconnection strategy other than NullReconnectionStrategy is specified.
  - Change the default types for callback and acceptable_replies in Connection._rpc
  - Validate the callback and acceptable_replies data types in Connection._rpc
- adapters.blocking_connection.BlockingConnection
  - Addition of _adapter_disconnect to blocking_connection.BlockingConnection
  - Add timeout methods to BlockingConnection addressing issue #41
  - BlockingConnection didn't allow you register more than one consumer callback because basic_consume was overridden to block immediately. New behavior allows you to do so.
  - Removed overriding of base basic_consume and basic_cancel methods. Now uses underlying Channel versions of those methods.
  - Added start_consuming() method to BlockingChannel to start the consumption loop.
  - Updated stop_consuming() to iterate through all the registered consumers in self._consumers and issue a basic_cancel.

0.9.14 - 2013-06-*

Bugfixes

- Major issue with socket buffer refactor in 0.9.13 (#328) fixes by cooper6581 and Erik Andersson
- Fix a bug in SelectConnection that did not allow for a IOLoop to be restarted (#337) fix by Ralf Nyrén
- Fix an issue in BlockingConnection disconnections (#340) fix by Mark Unsworth

Other

- Add NullHandler to prevent logging warnings when not configured (#339) by Cenk Altı
- Added Twisted Adapter example (#314) by nolinksoft

CHAPTER 4

Authors

- Tony Garnock-Jones
- Gavin M. Roy

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U