Pykka Documentation

Release 2.0.1

Stein Magnus Jodal

Oct 10, 2019

Usage

Pykka is a Python implementation of the actor model. The actor model introduces some simple rules to control the sharing of state and cooperation between execution units, which makes it easier to build concurrent applications.

For details and code examples, see the Pykka documentation.

Pykka is available from PyPI. To install it, run:

```
pip install pykka
```

Pykka works with CPython 2.7 and 3.5+, as well as PyPy 2.7 and 3.5+.

CHAPTER 1

Quickstart

Pykka is a Python implementation of the actor model. The actor model introduces some simple rules to control the sharing of state and cooperation between execution units, which makes it easier to build concurrent applications.

1.1 Rules of the actor model

- An actor is an execution unit that executes concurrently with other actors.
- An actor does not share state with anybody else, but it can have its own state.
- An actor can only communicate with other actors by sending and receiving messages. It can only send messages to actors whose address it has.
- When an actor receives a message it may take actions like:
  - altering its own state, e.g. so that it can react differently to a future message,
  - sending messages to other actors, or
  - starting new actors.

  None of the actions are required, and they may be applied in any order.
- An actor only processes one message at a time. In other words, a single actor does not give you any concurrency, and it does not need to use locks internally to protect its own state.

1.2 The actor implementations

Pykka's actor API comes with the following implementations:

- Threads: Each *`ThreadingActor`* is executed by a regular thread, i.e. `threading.Thread`. As handles for future results, it uses *`ThreadingFuture`* which is a thin wrapper around a `queue.Queue`. It has no dependencies outside Python itself. *`ThreadingActor`* plays well together with non-actor threads.

Note: If you monkey patch the standard library with `gevent` or `eventlet` you can still use *`ThreadingActor`* and *`ThreadingFuture`*. Python's threads will transparently use the underlying implementation provided by gevent or Eventlet.

- gevent: Each *`GeventActor`* is executed by a gevent greenlet. gevent is a coroutine-based Python networking library built on top of libev event loop. *`GeventActor`* is generally faster than *`ThreadingActor`*.
- Eventlet: Each *`EventletActor`* is executed by an Eventlet greenlet.

Pykka has an extensive test suite, and is tested on CPython 2.7, and 3.5+, as well as PyPy.

1.3 A basic actor

In its most basic form, a Pykka actor is a class with an *`on_receive()`* method:

```
import pykka

class Greeter(pykka.ThreadingActor):
    def on_receive(self, message):
        print('Hi there!')
```

To start an actor, you call the class' method *`start()`*, which starts the actor and returns an actor reference which can be used to communicate with the running actor:

```
actor_ref = Greeter.start()
```

If you need to pass arguments to the actor upon creation, you can pass them to the *`start()`* method, and receive them using the regular `__init__()` method:

```
import pykka

class Greeter(pykka.ThreadingActor):
    def __init__(self, greeting='Hi there!'):
        super().__init__()
        self.greeting = greeting

    def on_receive(self, message):
        print(self.greeting)

actor_ref = Greeter.start(greeting='Hi you!')
```

It can be useful to know that the init method is run in the execution context that starts the actor. There are also hooks for running code in the actor's own execution context when the actor starts, when it stops, and when an unhandled exception is raised. Check out the full API docs for the details.

To stop an actor, you can either call *`stop()`* on the *`ActorRef`*:

```
actor_ref.stop()
```

Or, if an actor wants to stop itself, it can simply do so:

```
self.stop()
```

Once an actor has been stopped, it cannot be restarted.

1.3.1 Sending messages

To send a message to the actor, you can either use the *tell()* method or the *ask()* method on the actor_ref object. *tell()* will fire off a message without waiting for an answer. In other words, it will never block. *ask()* will by default block until an answer is returned, potentially forever. If you provide a timeout keyword argument to *ask()*, you can specify for how long it should wait for an answer. If you want an answer, but don't need it right away because you have other stuff you can do first, you can pass block=False, and *ask()* will immediately return a "future" object.

The message itself can be of any type, for example a dict or your own message class type.

Summarized in code:

```
actor_ref.tell('Hi!')
# => Returns nothing. Will never block.

answer = actor_ref.ask('Hi?')
# => May block forever waiting for an answer

answer = actor_ref.ask('Hi?', timeout=3)
# => May wait 3s for an answer, then raises exception if no answer.

future = actor_ref.ask('Hi?', block=False)
# => Will return a future object immediately.
answer = future.get()
# => May block forever waiting for an answer
answer = future.get(timeout=0.1)
# => May wait 0.1s for an answer, then raises exception if no answer.
```

Warning: For performance reasons, Pykka **does not** clone the message you send before delivering it to the receiver. You are yourself responsible for either using immutable data structures or to copy.deepcopy() the data you're sending off to other actors.

1.3.2 Replying to messages

If a message is sent using actor_ref.ask() you can reply to the sender of the message by simply returning a value from the *on_receive()* method:

```
import pykka

class Greeter(pykka.ThreadingActor):
    def on_receive(self, message):
        return 'Hi there!'

actor_ref = Greeter.start()

answer = actor_ref.ask('Hi?')
print(answer)
# => 'Hi there!'
```

None is a valid response so if you return None explicitly, or don't return at all, a response containing None will be returned to the sender.

From the point of view of the actor it doesn't matter whether the message was sent using *tell()* or *ask()*. When the sender doesn't expect a response the *on_receive()* return value will be ignored.

The situation is similar in regard to exceptions: when *ask()* is used and you raise an exception from within *on_receive()* method, the exception will propagate to the sender:

```
import pykka

class Raiser(pykka.ThreadingActor):
    def on_receive(self, message):
        raise Exception('Oops')

actor_ref = Raiser.start()

try:
    actor_ref.ask('How are you?')
except Exception as e:
    print(repr(e))
    # => Exception('Oops')
```

1.4 Actor proxies

With the basic building blocks provided by actors and futures, we got everything we need to build more advanced abstractions. Pykka provides a single abstraction on top of the basic actor model, named "actor proxies". You can use Pykka without proxies, but we've found it to be a very convenient abstraction when building Mopidy.

Let's create an actor and start it:

```
import pykka

class Calculator(pykka.ThreadingActor):
    def __init__(self):
        super().__init__()
        self.last_result = None

    def add(self, a, b=None):
        if b is not None:
            self.last_result = a + b
        else:
            self.last_result += a
        return self.last_result

    def sub(self, a, b=None):
        if b is not None:
            self.last_result = a - b
        else:
            self.last_result -= a
        return self.last_result

actor_ref = Calculator.start()
```

You can create a proxy from any reference to a running actor:

```
proxy = actor_ref.proxy()
```

The proxy object will use introspection to figure out what public attributes and methods the actor has, and then mirror the full API of the actor. Any attribute or method prefixed with underscore will be ignored, which is the convention for keeping stuff private in Python.

When we access attributes or call methods on the proxy, it will ask the actor to access the given attribute or call the given method, and return the result to us. All results are wrapped in "future" objects, so you must use the *get()* method to get the actual data:

```
future = proxy.add(1, 3)
future.get()
# => 4

proxy.last_result.get()
# => 4
```

Since an actor only processes one message at the time and all messages are kept in order, you don't need to add the call to *get()* just to block processing until the actor has completed processing your last message:

```
proxy.sub(5)
proxy.add(3)
proxy.last_result.get()
# => 2
```

Since assignment doesn't return anything, it works just like on regular objects:

```
proxy.last_result = 17
proxy.last_result.get()
# => 17
```

Under the hood, the proxy does everything by sending messages to the actor using the regular *ask()* method we talked about previously. By doing so, it maintains the actor model restrictions. The only "magic" happening here is some basic introspection and automatic building of three different message types; one for method calls, one for attribute reads, and one for attribute writes.

1.4.1 Traversable attributes on proxies

Sometimes you'll want to access an actor attribute's methods or attributes through a proxy. For this case, Pykka supports "traversable attributes". By marking an actor attribute as traversable, Pykka will not return the attribute when accessed, but wrap it in a new proxy which is returned instead.

To mark an attribute as traversable, simply mark it with the *traversable()* function:

```
import pykka

class AnActor(pykka.ThreadingActor):
    playback = pykka.traversable(Playback())

class Playback(object):
    def play(self):
        return True

proxy = AnActor.start().proxy()
play_success = proxy.playback.play().get()
```

You can access methods and attributes nested as deep as you like, as long as all attributes on the path between the actor and the method or attribute on the end are marked as traversable.

CHAPTER 2

Examples

The `examples/` dir in Pykka's Git repo includes some runnable examples of Pykka usage.

2.1 Plain actor

```
#!/usr/bin/env python3

import pykka


GetMessages = object()


class PlainActor(pykka.ThreadingActor):
    def __init__(self):
        super().__init__()
        self.stored_messages = []

    def on_receive(self, message):
        if message is GetMessages:
            return self.stored_messages
        else:
            self.stored_messages.append(message)


if __name__ == '__main__':
    actor = PlainActor.start()
    actor.tell({'no': 'Norway', 'se': 'Sweden'})
    actor.tell({'a': 3, 'b': 4, 'c': 5})
    print(actor.ask(GetMessages))
    actor.stop()
```

Output:

```
[{'no': 'Norway', 'se': 'Sweden'}, {'a': 3, 'b': 4, 'c': 5}]
```

2.2 Actor with proxy

```
#!/usr/bin/env python3

import threading
import time

import pykka


class AnActor(pykka.ThreadingActor):
    field = 'this is the value of AnActor.field'

    def proc(self):
        log('this was printed by AnActor.proc()')

    def func(self):
        time.sleep(0.5)  # Block a bit to make it realistic
        return 'this was returned by AnActor.func() after a delay'


def log(msg):
    thread_name = threading.current_thread().name
    print(f'{thread_name}: {msg}')


if __name__ == '__main__':
    actor = AnActor.start().proxy()
    for i in range(3):
        # Method with side effect
        log('calling AnActor.proc() ...')
        actor.proc()

        # Method with return value
        log('calling AnActor.func() ...')
        result = actor.func()  # Does not block, returns a future
        log('printing result ... (blocking)')
        log(result.get())  # Blocks until ready

        # Field reading
        log('reading AnActor.field ...')
        result = actor.field  # Does not block, returns a future
        log('printing result ... (blocking)')
        log(result.get())  # Blocks until ready

        # Field writing
        log('writing AnActor.field ...')
        actor.field = 'new value'  # Assignment does not block
        result = actor.field  # Does not block, returns a future
        log('printing new field value ... (blocking)')
        log(result.get())  # Blocks until ready
    actor.stop()
```

Output:

```
MainThread: calling AnActor.proc() ...
MainThread: calling AnActor.func() ...
MainThread: printing result ... (blocking)
AnActor-1: this was printed by AnActor.proc()
MainThread: this was returned by AnActor.func() after a delay
MainThread: reading AnActor.field ...
MainThread: printing result ... (blocking)
MainThread: this is the value of AnActor.field
MainThread: writing AnActor.field ...
MainThread: printing new field value ... (blocking)
MainThread: new value
MainThread: calling AnActor.proc() ...
MainThread: calling AnActor.func() ...
MainThread: printing result ... (blocking)
AnActor-1: this was printed by AnActor.proc()
MainThread: this was returned by AnActor.func() after a delay
MainThread: reading AnActor.field ...
MainThread: printing result ... (blocking)
MainThread: new value
MainThread: writing AnActor.field ...
MainThread: printing new field value ... (blocking)
MainThread: new value
MainThread: calling AnActor.proc() ...
MainThread: calling AnActor.func() ...
AnActor-1: this was printed by AnActor.proc()
MainThread: printing result ... (blocking)
MainThread: this was returned by AnActor.func() after a delay
MainThread: reading AnActor.field ...
MainThread: printing result ... (blocking)
MainThread: new value
MainThread: writing AnActor.field ...
MainThread: printing new field value ... (blocking)
MainThread: new value
```

2.3 Multiple cooperating actors

```
#!/usr/bin/env python3

import pykka


class Adder(pykka.ThreadingActor):
    def add_one(self, i):
        print(f'{self} is increasing {i}')
        return i + 1


class Bookkeeper(pykka.ThreadingActor):
    def __init__(self, adder):
        super().__init__()
        self.adder = adder

    def count_to(self, target):
```

(continues on next page)

(continued from previous page)

```
        i = 0
        while i < target:
            i = self.adder.add_one(i).get()
            print(f'{self} got {i} back')


if __name__ == '__main__':
    adder = Adder.start().proxy()
    bookkeeper = Bookkeeper.start(adder).proxy()
    bookkeeper.count_to(10).get()
    pykka.ActorRegistry.stop_all()
```

Output:

```
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 0
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 1 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 1
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 2 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 2
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 3 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 3
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 4 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 4
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 5 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 5
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 6 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 6
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 7 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 7
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 8 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 8
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 9 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 9
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 10 back
```

2.4 Pool of actors sharing work

```
#!/usr/bin/env python3

"""
Resolve a bunch of IP addresses using a pool of resolver actors.

Based on example contributed by Kristian Klette <klette@klette.us>.

Either run without arguments:

    ./resolver.py

Or specify pool size and IPs to resolve:

    ./resolver.py 3 193.35.52.{1,2,3,4,5,6,7,8,9}
"""

import pprint
```

(continues on next page)

(continued from previous page)

```
import socket
import sys

import pykka


class Resolver(pykka.ThreadingActor):
    def resolve(self, ip):
        try:
            info = socket.gethostbyaddr(ip)
            print(f'Finished resolving {ip}')
            return info[0]
        except Exception:
            print(f'Failed resolving {ip}')
            return None


def run(pool_size, *ips):
    # Start resolvers
    resolvers = [Resolver.start().proxy() for _ in range(pool_size)]

    # Distribute work by mapping IPs to resolvers (not blocking)
    hosts = []
    for i, ip in enumerate(ips):
        hosts.append(resolvers[i % len(resolvers)].resolve(ip))

    # Gather results (blocking)
    ip_to_host = zip(ips, pykka.get_all(hosts))
    pprint.pprint(list(ip_to_host))

    # Clean up
    pykka.ActorRegistry.stop_all()


if __name__ == '__main__':
    if len(sys.argv[1:]) >= 2:
        run(int(sys.argv[1]), *sys.argv[2:])
    else:
        ips = [f'193.35.52.{i}' for i in range(1, 50)]
        run(10, *ips)
```

2.5 Mopidy music server

Pykka was originally created back in 2011 as a formalization of concurrency patterns that emerged in the Mopidy music server. The original Pykka source code wasn't extracted from Mopidy, but it built and improved on the concepts from Mopidy. Mopidy was later ported to build on Pykka instead of its own concurrency abstractions.

Mopidy still use Pykka extensively to keep independent parts, like the MPD and HTTP frontend servers or the Spotify and Google Music integrations, running independently. Every one of Mopidy's more than 100 extensions has at least one Pykka actor. By running each extension as an independent actor, errors and bugs in one extension is attempted isolated, to reduce the effect on the rest of the system.

You can browse the Mopidy source code to find many real life examples of Pykka usage.

CHAPTER 3

Pykka API

pykka.**__version__**
Pykka's **PEP 386** and **PEP 396** compatible version number

3.1 Actors

class pykka.**Actor**(**args*, ***kwargs*)
To create an actor:

1. subclass one of the *Actor* implementations:
   - *ThreadingActor*
   - *GeventActor*
   - *EventletActor*
2. implement your methods, including `__init__()`, as usual,
3. call *Actor.start()* on your actor class, passing the method any arguments for your constructor.

To stop an actor, call *Actor.stop()* or *ActorRef.stop()*.

For example:

```
import pykka

class MyActor(pykka.ThreadingActor):
    def __init__(self, my_arg=None):
        super().__init__()
        ... # My optional init code with access to start() arguments

    def on_start(self):
        ... # My optional setup code in same context as on_receive()

    def on_stop(self):
```

(continues on next page)

(continued from previous page)

```
        ... # My optional cleanup code in same context as on_receive()

    def on_failure(self, exception_type, exception_value, traceback):
        ... # My optional cleanup code in same context as on_receive()

    def on_receive(self, message):
        ... # My optional message handling code for a plain actor

    def a_method(self, ...):
        ... # My regular method to be used through an ActorProxy

my_actor_ref = MyActor.start(my_arg=...)
my_actor_ref.stop()
```

classmethod start(**args*, ***kwargs*)

Start an actor and register it in the *ActorRegistry*.

Any arguments passed to *start()* will be passed on to the class constructor.

Behind the scenes, the following is happening when you call *start()*:

1. The actor is created:
    1. *actor_urn* is initialized with the assigned URN.
    2. *actor_inbox* is initialized with a new actor inbox.
    3. *actor_ref* is initialized with a *pykka.ActorRef* object for safely communicating with the actor.
    4. At this point, your `__init__()` code can run.
2. The actor is registered in *pykka.ActorRegistry*.
3. The actor receive loop is started by the actor's associated thread/greenlet.

Returns a *ActorRef* which can be used to access the actor in a safe manner

actor_urn = None

The actor URN string is a universally unique identifier for the actor. It may be used for looking up a specific actor using *ActorRegistry.get_by_urn()*.

actor_inbox = None

The actor's inbox. Use *ActorRef.tell()*, *ActorRef.ask()*, and friends to put messages in the inbox.

actor_stopped = None

A `threading.Event` representing whether or not the actor should continue processing messages. Use *stop()* to change it.

actor_ref = None

The actor's *ActorRef* instance.

stop()

Stop the actor.

It's equivalent to calling *ActorRef.stop()* with `block=False`.

on_start()

Hook for doing any setup that should be done *after* the actor is started, but *before* it starts processing messages.

For *ThreadingActor*, this method is executed in the actor's own thread, while `__init__()` is executed in the thread that created the actor.

If an exception is raised by this method the stack trace will be logged, and the actor will stop.

on_stop()
Hook for doing any cleanup that should be done *after* the actor has processed the last message, and *before* the actor stops.

This hook is *not* called when the actor stops because of an unhandled exception. In that case, the *on_failure()* hook is called instead.

For *ThreadingActor* this method is executed in the actor's own thread, immediately before the thread exits.

If an exception is raised by this method the stack trace will be logged, and the actor will stop.

on_failure(*exception_type*, *exception_value*, *traceback*)
Hook for doing any cleanup *after* an unhandled exception is raised, and *before* the actor stops.

For *ThreadingActor* this method is executed in the actor's own thread, immediately before the thread exits.

The method's arguments are the relevant information from `sys.exc_info()`.

If an exception is raised by this method the stack trace will be logged, and the actor will stop.

on_receive(*message*)
May be implemented for the actor to handle regular non-proxy messages.

Parameters **message** (*any*) – the message to handle

Returns anything that should be sent as a reply to the sender

class pykka.**ActorRef**(*actor*)
Reference to a running actor which may safely be passed around.

ActorRef instances are returned by *Actor.start()* and the lookup methods in *ActorRegistry*. You should never need to create *ActorRef* instances yourself.

Parameters **actor** (*Actor*) – the actor to wrap

actor_class = None
The class of the referenced actor.

actor_urn = None
See *Actor.actor_urn*.

actor_inbox = None
See *Actor.actor_inbox*.

actor_stopped = None
See *Actor.actor_stopped*.

is_alive()
Check if actor is alive.

This is based on the actor's stopped flag. The actor is not guaranteed to be alive and responding even though *is_alive()* returns `True`.

Returns Returns `True` if actor is alive, `False` otherwise.

tell(*message*)
Send message to actor without waiting for any response.

Will generally not block, but if the underlying queue is full it will block until a free slot is available.

Parameters **message** (*any*) – message to send

Raise *pykka.ActorDeadError* if actor is not available

Returns nothing

ask(*message*, *block=True*, *timeout=None*)
Send message to actor and wait for the reply.

The message can be of any type. If `block` is `False`, it will immediately return a *Future* instead of blocking.

If `block` is `True`, and `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

Parameters

- **message** (*any*) – message to send
- **block** (*boolean*) – whether to block while waiting for a reply
- **timeout** (float or `None`) – seconds to wait before timeout if blocking

Raise *pykka.Timeout* if timeout is reached if blocking

Raise any exception returned by the receiving actor if blocking

Returns *pykka.Future*, or response if blocking

stop(*block=True*, *timeout=None*)
Send a message to the actor, asking it to stop.

Returns `True` if actor is stopped or was being stopped at the time of the call. `False` if actor was already dead. If `block` is `False`, it returns a future wrapping the result.

Messages sent to the actor before the actor is asked to stop will be processed normally before it stops.

Messages sent to the actor after the actor is asked to stop will be replied to with *pykka.ActorDeadError* after it stops.

The actor may not be restarted.

`block` and `timeout` works as for *ask()*.

Returns *pykka.Future*, or a boolean result if blocking

proxy()
Wraps the *ActorRef* in an *ActorProxy*.

Using this method like this:

```
proxy = AnActor.start().proxy()
```

is analogous to:

```
proxy = ActorProxy(AnActor.start())
```

Raise *pykka.ActorDeadError* if actor is not available

Returns *pykka.ActorProxy*

3.2 Proxies

class pykka.**ActorProxy**(*actor_ref*, *attr_path=None*)

An *ActorProxy* wraps an *ActorRef* instance. The proxy allows the referenced actor to be used through regular method calls and field access.

You can create an *ActorProxy* from any *ActorRef*:

```
actor_ref = MyActor.start()
actor_proxy = ActorProxy(actor_ref)
```

You can also get an *ActorProxy* by using *proxy()*:

```
actor_proxy = MyActor.start().proxy()
```

Attributes and method calls

When reading an attribute or getting a return value from a method, you get a *Future* object back. To get the enclosed value from the future, you must call *get()* on the returned future:

```
print(actor_proxy.string_attribute.get())
print(actor_proxy.count().get() + 1)
```

If you call a method just for it's side effects and do not care about the return value, you do not need to accept the returned future or call *get()* on the future. Simply call the method, and it will be executed concurrently with your own code:

```
actor_proxy.method_with_side_effect()
```

If you want to block your own code from continuing while the other method is processing, you can use *get()* to block until it completes:

```
actor_proxy.method_with_side_effect().get()
```

If you're using Python 3.5+, you can also use the `await` keyword to block until the method completes:

```
await actor_proxy.method_with_side_effect()
```

If you access a proxied method as an attribute, without calling it, you get an *CallableProxy*.

Proxy to itself

An actor can use a proxy to itself to schedule work for itself. The scheduled work will only be done after the current message and all messages already in the inbox are processed.

For example, if an actor can split a time consuming task into multiple parts, and after completing each part can ask itself to start on the next part using proxied calls or messages to itself, it can react faster to other incoming messages as they will be interleaved with the parts of the time consuming task. This is especially useful for being able to stop the actor in the middle of a time consuming task.

To create a proxy to yourself, use the actor's *actor_ref* attribute:

```
proxy_to_myself_in_the_future = self.actor_ref.proxy()
```

If you create a proxy in your actor's constructor or *on_start* method, you can create a nice API for deferring work to yourself in the future:

```
def __init__(self):
    ...
    self._in_future = self.actor_ref.proxy()
    ...

def do_work(self):
    ...
    self._in_future.do_more_work()
    ...

def do_more_work(self):
    ...
```

To avoid infinite loops during proxy introspection, proxies to self should be kept as private instance attributes by prefixing the attribute name with _.

Examples

An example of *ActorProxy* usage:

```
#!/usr/bin/env python3

import pykka


class Adder(pykka.ThreadingActor):
    def add_one(self, i):
        print(f'{self} is increasing {i}')
        return i + 1


class Bookkeeper(pykka.ThreadingActor):
    def __init__(self, adder):
        super().__init__()
        self.adder = adder

    def count_to(self, target):
        i = 0
        while i < target:
            i = self.adder.add_one(i).get()
            print(f'{self} got {i} back')


if __name__ == '__main__':
    adder = Adder.start().proxy()
    bookkeeper = Bookkeeper.start(adder).proxy()
    bookkeeper.count_to(10).get()
    pykka.ActorRegistry.stop_all()
```

Parameters **actor_ref** (*pykka.ActorRef*) – reference to the actor to proxy

Raise *pykka.ActorDeadError* if actor is not available

actor_ref = None
The actor's *pykka.ActorRef* instance.

class pykka.**CallableProxy**(*actor_ref*, *attr_path*)
Proxy to a single method.

CallableProxy instances are returned when accessing methods on a *ActorProxy* without calling them.

Example:

```
proxy = AnActor.start().proxy()

# Ask semantics returns a future. See `__call__()` docs.
future = proxy.do_work()

# Tell semantics are fire and forget. See `defer()` docs.
proxy.do_work.defer()
```

__call__(**args*, ***kwargs*)
Call with *ask()* semantics.

Returns a future which will yield the called method's return value.

If the call raises an exception is set on the future, and will be reraised by *get()*. If the future is left unused, the exception will not be reraised. Either way, the exception will also be logged. See *Logging* for details.

defer(**args*, ***kwargs*)
Call with *tell()* semantics.

Does not create or return a future.

If the call raises an exception, there is no future to set the exception on. Thus, the actor's *on_failure()* hook is called instead.

New in version 2.0.

pykka.**traversable**(*obj*)
Marks an actor attribute as traversable.

The traversable marker makes the actor attribute's own methods and attributes available to users of the actor through an *ActorProxy*.

Used as a function to mark a single attribute:

```
class AnActor(pykka.ThreadingActor):
    playback = pykka.traversable(Playback())

class Playback(object):
    def play(self):
        return True
```

This function can also be used as a class decorator, making all instances of the class traversable:

```
class AnActor(pykka.ThreadingActor):
    playback = Playback()

@pykka.traversable
class Playback(object):
    def play(self):
        return True
```

The third alternative, and the only way in Pykka < 2.0, is to manually mark a class as traversable by setting the `pykka_traversable` attribute to `True`:

```
class AnActor(pykka.ThreadingActor):
    playback = Playback()
```

(continues on next page)

(continued from previous page)

```
class Playback(object):
    pykka_traversable = True

    def play(self):
        return True
```

When the attribute is marked as traversable, its methods can be executed in the context of the actor through an actor proxy:

```
proxy = AnActor.start().proxy()
assert proxy.playback.play().get() is True
```

New in version 2.0.

3.3 Futures

class pykka.**Future**

A *Future* is a handle to a value which is available or will be available in the future.

Typically returned by calls to actor methods or accesses to actor fields.

To get hold of the encapsulated value, call *Future.get()* or, if using Python 3.5+, `await` the future.

get(*timeout=None*)

Get the value encapsulated by the future.

If the encapsulated value is an exception, it is raised instead of returned.

If `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

The encapsulated value can be retrieved multiple times. The future will only block the first time the value is accessed.

Parameters **timeout** (float or `None`) – seconds to wait before timeout

Raise *pykka.Timeout* if timeout is reached

Raise encapsulated value if it is an exception

Returns encapsulated value if it is not an exception

set(*value=None*)

Set the encapsulated value.

Parameters **value** (any object or `None`) – the encapsulated value or nothing

Raise an exception if set is called multiple times

set_exception(*exc_info=None*)

Set an exception as the encapsulated value.

You can pass an `exc_info` three-tuple, as returned by `sys.exc_info()`. If you don't pass `exc_info`, `sys.exc_info()` will be called and the value returned by it used.

In other words, if you're calling *set_exception()*, without any arguments, from an except block, the exception you're currently handling will automatically be set on the future.

Parameters **exc_info** – *(three-tuple of (exc_class, exc_instance, traceback))* – the encapsulated exception

set_get_hook (*func*)

Set a function to be executed when *get()* is called.

The function will be called when *get()* is called, with the `timeout` value as the only argument. The function's return value will be returned from *get()*.

New in version 1.2.

Parameters **func** (*function accepting a timeout value*) – called to produce return value of *get()*

filter (*func*)

Return a new future with only the items passing the predicate function.

If the future's value is an iterable, *filter()* will return a new future whose value is another iterable with only the items from the first iterable for which `func(item)` is true. If the future's value isn't an iterable, a `TypeError` will be raised when *get()* is called.

Example:

```
>>> import pykka
>>> f = pykka.ThreadingFuture()
>>> g = f.filter(lambda x: x > 10)
>>> g
<pykka.future.ThreadingFuture at ...>
>>> f.set(range(5, 15))
>>> f.get()
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> g.get()
[11, 12, 13, 14]
```

New in version 1.2.

join (**futures*)

Return a new future with a list of the result of multiple futures.

One or more futures can be passed as arguments to *join()*. The new future returns a list with the results from all the joined futures.

Example:

```
>>> import pykka
>>> a = pykka.ThreadingFuture()
>>> b = pykka.ThreadingFuture()
>>> c = pykka.ThreadingFuture()
>>> f = a.join(b, c)
>>> a.set('def')
>>> b.set(123)
>>> c.set(False)
>>> f.get()
['def', 123, False]
```

New in version 1.2.

map (*func*)

Return a new future with the result of the future passed through a function.

Example:

```
>>> import pykka
>>> f = pykka.ThreadingFuture()
>>> g = f.map(lambda x: x + 10)
>>> f.set(30)
>>> g.get()
40

>>> f = pykka.ThreadingFuture()
>>> g = f.map(lambda x: x['foo'])
>>> f.set({'foo': 'bar'})
>>> g.get()
'bar'
```

New in version 1.2.

Changed in version 2.0: Previously, if the future's result was an iterable (except a string), the function was applied to each item in the iterable. This behavior is unpredictable and makes regular use cases like extracting a single field from a dict difficult, thus the behavior has been simplified. Now, the entire result value is passed to the function.

reduce (*func*[, *initial*])

Return a new future with the result of reducing the future's iterable into a single value.

The function of two arguments is applied cumulatively to the items of the iterable, from left to right. The result of the first function call is used as the first argument to the second function call, and so on, until the end of the iterable. If the future's value isn't an iterable, a `TypeError` is raised.

`reduce()` accepts an optional second argument, which will be used as an initial value in the first function call. If the iterable is empty, the initial value is returned.

Example:

```
>>> import pykka
>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y)
>>> f.set(['a', 'b', 'c'])
>>> g.get()
'abc'

>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y)
>>> f.set([1, 2, 3])
>>> (1 + 2) + 3
6
>>> g.get()
6

>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y, 5)
>>> f.set([1, 2, 3])
>>> ((5 + 1) + 2) + 3
11
>>> g.get()
11

>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y, 5)
>>> f.set([])
```

(continues on next page)

(continued from previous page)

```
>>> g.get()
5
```

New in version 1.2.

pykka.**get_all**(*futures*, *timeout=None*)

Collect all values encapsulated in the list of futures.

If `timeout` is not `None`, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

Parameters

- **futures** (list of *pykka.Future*) – futures for the results to collect
- **timeout** (float or `None`) – seconds to wait before timeout

Raise *pykka.Timeout* if timeout is reached

Returns list of results

3.4 Registry

class pykka.**ActorRegistry**

Registry which provides easy access to all running actors.

Contains global state, but should be thread-safe.

classmethod broadcast(*message*, *target_class=None*)

Broadcast `message` to all actors of the specified `target_class`.

If no `target_class` is specified, the message is broadcasted to all actors.

Parameters

- **message** (*any*) – the message to send
- **target_class** (*class or class name*) – optional actor class to broadcast the message to

classmethod get_all()

Get *ActorRef* for all running actors.

Returns list of *pykka.ActorRef*

classmethod get_by_class(*actor_class*)

Get *ActorRef* for all running actors of the given class, or of any subclass of the given class.

Parameters **actor_class** (*class*) – actor class, or any superclass of the actor

Returns list of *pykka.ActorRef*

classmethod get_by_class_name(*actor_class_name*)

Get *ActorRef* for all running actors of the given class name.

Parameters **actor_class_name** (*string*) – actor class name

Returns list of *pykka.ActorRef*

classmethod get_by_urn(*actor_urn*)

Get an actor by its universally unique URN.

Parameters **actor_urn** (*string*) – actor URN

Returns *pykka.ActorRef* or `None` if not found

classmethod register(*actor_ref*)
Register an *ActorRef* in the registry.

This is done automatically when an actor is started, e.g. by calling *Actor.start()*.

Parameters **actor_ref** (*pykka.ActorRef*) – reference to the actor to register

classmethod stop_all(*block=True*, *timeout=None*)
Stop all running actors.

`block` and `timeout` works as for *ActorRef.stop()*.

If `block` is `True`, the actors are guaranteed to be stopped in the reverse of the order they were started in. This is helpful if you have simple dependencies in between your actors, where it is sufficient to shut down actors in a LIFO manner: last started, first stopped.

If you have more complex dependencies in between your actors, you should take care to shut them down in the required order yourself, e.g. by stopping dependees from a dependency's *on_stop()* method.

Returns If not blocking, a list with a future for each stop action. If blocking, a list of return values from *pykka.ActorRef.stop()*.

classmethod unregister(*actor_ref*)
Remove an *ActorRef* from the registry.

This is done automatically when an actor is stopped, e.g. by calling *Actor.stop()*.

Parameters **actor_ref** (*pykka.ActorRef*) – reference to the actor to unregister

3.5 Exceptions

exception pykka.**ActorDeadError**
Exception raised when trying to use a dead or unavailable actor.

exception pykka.**Timeout**
Exception raised at future timeout.

3.6 Messages

The *pykka.messages* module contains Pykka's own actor messages.

In general, you should not need to use any of these classes. However, they have been made part of the public API so that certain optimizations can be done without touching Pykka's internals.

An example is to combine *ask()* and *ProxyCall* to call a method on an actor without having to spend any resources on creating a proxy object:

```
reply = actor_ref.ask(
    ProxyCall(
        attr_path=['my_method'],
        args=['foo'],
        kwargs={'bar': 'baz'}
    )
)
```

Another example is to use *tell()* instead of *ask()* for the proxy method call, and thus avoid the creation of a future for the return value if you don't need it.

It should be noted that these optimizations should only be necessary in very special circumstances.

New in version 2.0.

class pykka.messages.**ProxyCall**(*attr_path*, *args*, *kwargs*)

args
Alias for field number 1

attr_path
Alias for field number 0

kwargs
Alias for field number 2

class pykka.messages.**ProxyGetAttr**(*attr_path*)

attr_path
Alias for field number 0

class pykka.messages.**ProxySetAttr**(*attr_path*, *value*)

attr_path
Alias for field number 0

value
Alias for field number 1

3.7 Logging

Pykka uses Python's standard `logging` module for logging debug messages and any unhandled exceptions in the actors. All log messages emitted by Pykka are issued to the logger named `pykka`, or a sub-logger of it.

3.7.1 Log levels

Pykka logs at several different log levels, so that you can filter out the parts you're not interested in:

CRITICAL (highest) This level is only used by the debug helpers in *pykka.debug*.

ERROR Exceptions raised by an actor that are not captured into a reply future are logged at this level.

WARNING Unhandled messages and other potential programming errors are logged at this level.

INFO Exceptions raised by an actor that are captured into a reply future are logged at this level. If the future result is used elsewhere, the exceptions is reraised there too. If the future result isn't used, the log message is the only trace of the exception happening.

To catch bugs earlier, it is recommended to show log messages this level during development.

DEBUG (lowest) Every time an actor is started or stopped, and registered or unregistered in the actor registry, a message is logged at this level.

In summary, you probably want to always let log messages at `WARNING` and higher through, while `INFO` should also be kept on during development.

3.7.2 Log handlers

Out of the box, Pykka is set up with `logging.NullHandler` as the only log record handler. This is the recommended approach for logging in libraries, so that the application developer using the library will have full control over how the log messages from the library will be exposed to the application's users.

In other words, if you want to see the log messages from Pykka anywhere, you need to add a useful handler to the root logger or the logger named `pykka` to get any log output from Pykka.

The defaults provided by `logging.basicConfig()` is enough to get debug log messages from Pykka:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

3.7.3 Recommended setup

If your application is already using `logging`, and you want debug log output from your own application, but not from Pykka, you can ignore debug log messages from Pykka by increasing the threshold on the Pykka logger to `INFO` level or higher:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('pykka').setLevel(logging.INFO)
```

Given that you've fixed all unhandled exceptions logged at the `INFO` level during development, you probably want to disable logging from Pykka at the `INFO` level in production to avoid logging exceptions that are properly handled:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('pykka').setLevel(logging.WARNING)
```

For more details on how to use `logging`, please refer to the Python standard library documentation.

3.8 Debug helpers

pykka.debug.**log_thread_tracebacks**(**args*, ***kwargs*)

Logs at `logging.CRITICAL` level a traceback for each running thread.

This can be a convenient tool for debugging deadlocks.

The function accepts any arguments so that it can easily be used as e.g. a signal handler, but it does not use the arguments for anything.

To use this function as a signal handler, setup logging with a `logging.CRITICAL` threshold or lower and make your main thread register this with the `signal` module:

```
import logging
import signal

import pykka.debug

logging.basicConfig(level=logging.DEBUG)
signal.signal(signal.SIGUSR1, pykka.debug.log_thread_tracebacks)
```

If your application deadlocks, send the *SIGUSR1* signal to the process:

```
kill -SIGUSR1 <pid of your process>
```

Signal handler caveats:

- The function *must* be registered as a signal handler by your main thread. If not, `signal.signal()` will raise a `ValueError`.
- All signals in Python are handled by the main thread. Thus, the signal will only be handled, and the tracebacks logged, if your main thread is available to do some work. Making your main thread idle using `time.sleep()` is OK. The signal will awaken your main thread. Blocking your main thread on e.g. `queue.Queue.get()` or *`pykka.Future.get()`* will break signal handling, and thus you won't be able to signal your process to print the thread tracebacks.

The morale is: setup signals using your main thread, start your actors, then let your main thread relax for the rest of your application's life cycle.

New in version 1.1.

3.8.1 Deadlock debugging

This is a complete example of how to use *`log_thread_tracebacks()`* to debug deadlocks:

```
#!/usr/bin/env python3

import logging
import os
import signal
import time

import pykka
import pykka.debug


class DeadlockActorA(pykka.ThreadingActor):
    def foo(self, b):
        logging.debug('This is foo calling bar')
        return b.bar().get()


class DeadlockActorB(pykka.ThreadingActor):
    def __init__(self, a):
        super().__init__()
        self.a = a

    def bar(self):
        logging.debug('This is bar calling foo; BOOM!')
        return self.a.foo().get()


if __name__ == '__main__':
    print('Setting up logging to get output from signal handler...')
    logging.basicConfig(level=logging.DEBUG)

    print('Registering signal handler...')
    signal.signal(signal.SIGUSR1, pykka.debug.log_thread_tracebacks)

    print('Starting actors...')
```

(continues on next page)

(continued from previous page)

```
    a = DeadlockActorA.start().proxy()
    b = DeadlockActorB.start(a).proxy()

    print('Now doing something stupid that will deadlock the actors...')
    a.foo(b)

    time.sleep(0.01)  # Yield to actors, so we get output in a readable order

    pid = os.getpid()
    print('Making main thread relax; not block, not quit')
    print('1) Use `kill -SIGUSR1 {:d}` to log thread tracebacks'.format(pid))
    print('2) Then `kill {:d}` to terminate the process'.format(pid))
    while True:
        time.sleep(1)
```

Running the script outputs the following:

```
Setting up logging to get output from signal handler...
Registering signal handler...
Starting actors...
DEBUG:pykka:Registered DeadlockActorA (urn:uuid:60803d09-cf5a-46cc-afdc-0c813e2e6647)
DEBUG:pykka:Starting DeadlockActorA (urn:uuid:60803d09-cf5a-46cc-afdc-0c813e2e6647)
DEBUG:pykka:Registered DeadlockActorB (urn:uuid:626adc83-ae35-439c-866a-85a3e29fd42c)
DEBUG:pykka:Starting DeadlockActorB (urn:uuid:626adc83-ae35-439c-866a-85a3e29fd42c)
Now doing something stupid that will deadlock the actors...
DEBUG:root:This is foo calling bar
DEBUG:root:This is bar calling foo; BOOM!
Making main thread relax; not block, not quit
1) Use `kill -SIGUSR1 2284` to log thread tracebacks
2) Then `kill 2284` to terminate the process
```

The two actors are now deadlocked waiting for each other while the main thread is idling, ready to process any signals.

To debug the deadlock, send the `SIGUSR1` signal to the process, which has PID 2284 in this example:

```
kill -SIGUSR1 2284
```

This makes the main thread log the current traceback for each thread. The logging output shows that the two actors are both waiting for data from the other actor:

```
CRITICAL:pykka:Current state of DeadlockActorB-2 (ident: 140151493752576):
File "/usr/lib/python3.6/threading.py", line 884, in _bootstrap
    self._bootstrap_inner()
File "/usr/lib/python3.6/threading.py", line 916, in _bootstrap_inner
    self.run()
File "/usr/lib/python3.6/threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
File ".../pykka/actor.py", line 195, in _actor_loop
    response = self._handle_receive(message)
File ".../pykka/actor.py", line 297, in _handle_receive
    return callee(*message['args'], **message['kwargs'])
File "examples/deadlock_debugging.py", line 25, in bar
    return self.a.foo().get()
File ".../pykka/threading.py", line 47, in get
    self._data = self._queue.get(True, timeout)
File "/usr/lib/python3.6/queue.py", line 164, in get
    self.not_empty.wait()
```

(continues on next page)

(continued from previous page)

```
File "/usr/lib/python3.6/threading.py", line 295, in wait
    waiter.acquire()

CRITICAL:pykka:Current state of DeadlockActorA-1 (ident: 140151572883200):
File "/usr/lib/python3.6/threading.py", line 884, in _bootstrap
    self._bootstrap_inner()
File "/usr/lib/python3.6/threading.py", line 916, in _bootstrap_inner
    self.run()
File "/usr/lib/python3.6/threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
File ".../pykka/actor.py", line 195, in _actor_loop
    response = self._handle_receive(message)
File ".../pykka/actor.py", line 297, in _handle_receive
    return callee(*message['args'], **message['kwargs'])
File "examples/deadlock_debugging.py", line 15, in foo
    return b.bar().get()
File ".../pykka/threading.py", line 47, in get
    self._data = self._queue.get(True, timeout)
File "/usr/lib/python3.6/queue.py", line 164, in get
    self.not_empty.wait()
File "/usr/lib/python3.6/threading.py", line 295, in wait
    waiter.acquire()

CRITICAL:pykka:Current state of MainThread (ident: 140151593330496):
File ".../examples/deadlock_debugging.py", line 49, in <module>
    time.sleep(1)
File ".../pykka/debug.py", line 63, in log_thread_tracebacks
    stack = ''.join(traceback.format_stack(frame))
```

CHAPTER 4

Runtimes

By default, Pykka builds on top of Python's regular threading concurrency model, via the standard library modules `threading` and `queue`.

Alternatively, you may run Pykka on top of `gevent` or `eventlet`.

Note that Pykka does no attempt at supporting a mix of concurrency runtimes. Such a future feature has briefly been discussed in issue #11.

4.1 Threading

4.1.1 Installation

The default threading runtime has no dependencies other than Pykka itself and the Python standard library.

4.1.2 API

class pykka.**ThreadingFuture**

ThreadingFuture implements *Future* for use with *ThreadingActor*.

The future is implemented using a `queue.Queue`.

The future does *not* make a copy of the object which is *set()* on it. It is the setters responsibility to only pass immutable objects or make a copy of the object before setting it on the future.

Changed in version 0.14: Previously, the encapsulated value was a copy made with `copy.deepcopy()`, unless the encapsulated value was a future, in which case the original future was encapsulated.

get(*timeout=None*)

Get the value encapsulated by the future.

If the encapsulated value is an exception, it is raised instead of returned.

If `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

The encapsulated value can be retrieved multiple times. The future will only block the first time the value is accessed.

Parameters **timeout** (float or `None`) – seconds to wait before timeout

Raise *pykka.Timeout* if timeout is reached

Raise encapsulated value if it is an exception

Returns encapsulated value if it is not an exception

set(*value=None*)

Set the encapsulated value.

Parameters **value** (any object or `None`) – the encapsulated value or nothing

Raise an exception if set is called multiple times

set_exception(*exc_info=None*)

Set an exception as the encapsulated value.

You can pass an `exc_info` three-tuple, as returned by `sys.exc_info()`. If you don't pass `exc_info`, `sys.exc_info()` will be called and the value returned by it used.

In other words, if you're calling *set_exception()*, without any arguments, from an except block, the exception you're currently handling will automatically be set on the future.

Parameters **exc_info** (*three-tuple of (exc_class, exc_instance, traceback)*) – the encapsulated exception

class pykka.**ThreadingActor**(**args*, ***kwargs*)

ThreadingActor implements *Actor* using regular Python threads.

This implementation is slower than *GeventActor*, but can be used in a process with other threads that are not Pykka actors.

use_daemon_thread = False

A boolean value indicating whether this actor is executed on a thread that is a daemon thread (`True`) or not (`False`). This must be set before *pykka.Actor.start()* is called, otherwise `RuntimeError` is raised.

The entire Python program exits when no alive non-daemon threads are left. This means that an actor running on a daemon thread may be interrupted at any time, and there is no guarantee that cleanup will be done or that *pykka.Actor.on_stop()* will be called.

Actors do not inherit the daemon flag from the actor that made it. It always has to be set explicitly for the actor to run on a daemonic thread.

4.2 gevent

4.2.1 Installation

To run Pykka on top of gevent, you first need to install the gevent package from PyPI:

```
pip install gevent
```

4.2.2 Code changes

Next, all actors must subclass *pykka.gevent.GeventActor* instead of *pykka.ThreadingActor*.

If you create any futures yourself, you must replace *pykka.ThreadingFuture* with *pykka.gevent.GeventFuture*.

With those changes in place, Pykka should run on top of gevent.

4.2.3 API

class pykka.gevent.**GeventFuture**(*async_result=None*)

GeventFuture implements *pykka.Future* for use with *GeventActor*.

It encapsulates a `gevent.event.AsyncResult` object which may be used directly, though it will couple your code with gevent.

async_result = None

The encapsulated `gevent.event.AsyncResult`

get(*timeout=None*)

Get the value encapsulated by the future.

If the encapsulated value is an exception, it is raised instead of returned.

If `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

The encapsulated value can be retrieved multiple times. The future will only block the first time the value is accessed.

Parameters **timeout** (float or `None`) – seconds to wait before timeout

Raise *pykka.Timeout* if timeout is reached

Raise encapsulated value if it is an exception

Returns encapsulated value if it is not an exception

set(*value=None*)

Set the encapsulated value.

Parameters **value** (any object or `None`) – the encapsulated value or nothing

Raise an exception if set is called multiple times

set_exception(*exc_info=None*)

Set an exception as the encapsulated value.

You can pass an `exc_info` three-tuple, as returned by `sys.exc_info()`. If you don't pass `exc_info`, `sys.exc_info()` will be called and the value returned by it used.

In other words, if you're calling *set_exception()*, without any arguments, from an except block, the exception you're currently handling will automatically be set on the future.

Parameters **exc_info** (*three-tuple of (exc_class, exc_instance, traceback)*) – the encapsulated exception

class pykka.gevent.**GeventActor**(**args*, ***kwargs*)

GeventActor implements *pykka.Actor* using the gevent library. gevent is a coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of libevent event loop.

This is a very fast implementation.

4.3 eventlet

4.3.1 Installation

To run Pykka on top of eventlet, you first need to install the eventlet package from PyPI:

```
pip install eventlet
```

4.3.2 Code changes

Next, all actors must subclass *pykka.eventlet.EventletActor* instead of *pykka.ThreadingActor*.

If you create any futures yourself, you must replace *pykka.ThreadingFuture* with *pykka.eventlet.EventletFuture*.

With those changes in place, Pykka should run on top of eventlet.

4.3.3 API

class pykka.eventlet.**EventletEvent**

EventletEvent adapts `eventlet.event.Event` to `threading.Event` interface.

class pykka.eventlet.**EventletFuture**

EventletFuture implements *pykka.Future* for use with *EventletActor*.

get(*timeout=None*)

Get the value encapsulated by the future.

If the encapsulated value is an exception, it is raised instead of returned.

If `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

The encapsulated value can be retrieved multiple times. The future will only block the first time the value is accessed.

Parameters **timeout** (float or `None`) – seconds to wait before timeout

Raise *pykka.Timeout* if timeout is reached

Raise encapsulated value if it is an exception

Returns encapsulated value if it is not an exception

set(*value=None*)

Set the encapsulated value.

Parameters **value** (any object or `None`) – the encapsulated value or nothing

Raise an exception if set is called multiple times

set_exception(*exc_info=None*)

Set an exception as the encapsulated value.

You can pass an `exc_info` three-tuple, as returned by `sys.exc_info()`. If you don't pass `exc_info`, `sys.exc_info()` will be called and the value returned by it used.

In other words, if you're calling *set_exception()*, without any arguments, from an except block, the exception you're currently handling will automatically be set on the future.

Parameters **exc_info** – (*three-tuple of (exc_class, exc_instance, traceback)*) – the encapsulated exception

class pykka.eventlet.**EventletActor**(**args*, ***kwargs*)

EventletActor implements *pykka.Actor* using the eventlet library.

This implementation uses eventlet green threads.

CHAPTER 5

Testing

Pykka actors can be tested using the regular Python testing tools like pytest, `unittest`, and `unittest.mock`.

To test actors in a setting as close to production as possible, a typical pattern is the following:

1. In the test setup, start an actor together with any actors/collaborators it depends on. The dependencies will often be replaced by mocks to control their behavior.
2. In the test, *`ask()`* or *`tell()`* the actor something.
3. In the test, assert on the actor's state or the return value from the *`ask()`*.
4. In the test teardown, stop the actor to properly clean up before the next test.

5.1 An example

Let's look at an example actor that we want to test:

```
import pykka


class ProducerActor(pykka.ThreadingActor):
    def __init__(self, consumer):
        super(ProducerActor, self).__init__()
        self.consumer = consumer

    def produce(self):
        new_item = {'item': 1, 'new': True}
        self.consumer.consume(new_item)
```

We can test this actor with pytest by mocking the consumer and asserting that it receives a newly produced item:

```
from producer import ProducerActor
```

(continues on next page)

(continued from previous page)

```
import pytest


@pytest.fixture
def consumer_mock(mocker):
    yield mocker.Mock()


@pytest.fixture
def producer(consumer_mock):
    # Step 1: The actor under test is wired up with
    # its dependencies and is started.
    proxy = ProducerActor.start(consumer_mock).proxy()

    yield proxy

    # Step 4: The actor is stopped to clean up before the next test.
    proxy.stop()


def test_producer_actor(consumer_mock, producer):
    # Step 2: Interact with the actor.
    # We call .get() on the last future returned by the actor to wait
    # for the actor to process all messages before asserting anything.
    producer.produce().get()

    # Step 3: Assert that the return values or actor state is as expected.
    consumer_mock.consume.assert_called_once_with({'item': 1, 'new': True})
```

If this way of setting up and tearing down test resources is unfamiliar to you, it is strongly recommended to read up on pytest's great fixture feature.

CHAPTER 6

Inspiration

Much of the naming of concepts and methods in Pykka is taken from the Akka project which implements actors on the JVM. Though, Pykka does not aim to be a Python port of Akka, and supports far fewer features.

6.1 What Pykka is not

Notably, Pykka **does not** support the following features:

- Supervision: Linking actors, supervisors, or supervisor groups.
- Remoting: Communicating with actors running on other hosts.
- Routers: Pykka does not come with a set of predefined message routers, though you may make your own actors for routing messages.

CHAPTER 7

Changes

7.1 v2.0.1 (2019-10-10)

Bugfix release.

- Make *ActorRef* hashable.

7.2 v2.0.0 (2019-05-07)

Major feature release.

7.2.1 Dependencies

- Drop support for Python 2.6, 3.2, 3.3, and 3.4. All have reached their end of life and do no longer receive security updates.
- Include CPython 3.5, 3.6, 3.7, and 3.8 pre-releases, and PyPy 3.5 in the test matrix.
- Include gevent and Eventlet tests in all environments. Since Pykka was originally developed, both has grown support for Python 3 and PyPy.
- On Python 3, import `Callable` and `Iterable` from `collections.abc` instead of `collections`. This fixes a deprecation warning on Python 3.7 and prepares for Python 3.8.

7.2.2 Actors

- Actor messages are no longer required to be `dict` objects. Any object type can be used as an actor message. (Fixes: #39, #45, PR: #79)

  For existing code, this means that *on_receive()* implementations should no longer assume the received message to be a `dict`, and guard with the appropriate amount of `isinstance()` checks. As an existing

application will not observe any new message types before it starts using them itself, this is not marked as backwards incompatible.

7.2.3 Proxies

- **Backwards incompatible:** Avoid accessing actor properties when creating a proxy for the actor. For properties with side effects, this is a major bug fix. For properties which does heavy work, this is a major startup performance improvement.

  This is backwards incompatible if you in a property getter returned an object instance with the `pykka_traversable` marker. Previously, this would work just like a traversable attribute. Now, the property always returns a future with the property getter's return value.

- Fix infinite recursion when creating a proxy for an actor with an attribute or method replaced with a `Mock` without a `spec` defined. (Fixes: #26, #27)

- Fix infinite recursion when creating a proxy for an actor with an attribute that was itself a proxy to the same actor. The attribute will now be ignored and a warning log message will ask you to consider making the self-proxy private. (Fixes: #48)

- Add *`defer()`* to support method calls through a proxy with *`tell()`* semantics. (Contributed by Andrey Gubarev. Fixes: #63. PR: #72)

- Add *`traversable()`* for marking an actor's attributes as traversable when used through actor proxies. The old way of manually adding a `pykka_traversable` attribute to the object to be traversed still works, but the new function is recommended as it provides protection against typos in the marker name, and keeps the traversable marking in the actor class itself. (PR: #81)

7.2.4 Futures

- **Backwards incompatible:** *`pykka.Future.set_exception()`* no longer accepts an exception instance, which was deprecated in 0.15. The method can be called with either an `exc_info` tuple or `None`, in which case it will use `sys.exc_info()` to get information on the current exception.

- **Backwards incompatible:** *`pykka.Future.map()`* on a future with an iterable result no longer applies the map function to each item in iterable. Instead, the entire future result is passed to the map function. (Fixes: #64)

  To upgrade existing code, make sure to explicitly apply the core of your map function to each item in the iterable:

```
>>> f = pykka.ThreadingFuture()
>>> f.set([1, 2, 3])
>>> f.map(lambda x: x + 1).get()  # Pykka < 2.0
[2, 3, 4]
>>> f.map(lambda x: [i + 1 for i in x]).get()  # Pykka >= 2.0
[2, 3, 4]
```

  This change makes it easy to use *`map()`* to extract a field from a future that returns a dict:

```
>>> f = pykka.ThreadingFuture()
>>> f.set({'foo': 'bar'})
>>> f.map(lambda x: x['foo']).get()
'bar'
```

  Because dict is an iterable, the now removed special handling of iterables made this pattern difficult to use.

- Reuse result from *`pykka.Future.filter()`*, *`pykka.Future.map()`*, and *`pykka.Future.reduce()`*. Recalculating the result on each call to *`pykka.Future.get()`* is both inconsistent with regular futures and can cause problems if the function is expensive or has side effects. (Fixes: #32)

- If using Python 3.5+, one can now use the `await` keyword to get the result from a future. (Contributed by Joshua Doncaster-Marsiglio. PR: #78)

7.2.5 Logging

- Pykka's use of different log levels has been *documented*.
- Exceptions raised by an actor that are captured into a reply future are now logged on the `INFO` level instead of the `DEBUG` level. This makes it possible to detect potentially unhandled exceptions during development without having to turn on debug logging, which can have a low signal to noise ratio. (Contributed by Stefan Möhl. Fixes: #73)

7.2.6 Gevent support

- Ensure that the original traceback is preserved when an exception is returned through a future from a Gevent actor. (Contributed by Arne Brutschy. Fixes: #74, PR: #75)

7.2.7 Internals

- **Backwards incompatible:** Prefix all internal modules with _. This is backwards incompatible if you have imported objects from other import paths than what is used in the documentation.
- Port tests to pytest.
- Format code with Black.
- Change internal messaging format from `dict` to `namedtuple`. (PR: #80)

7.3 v1.2.1 (2015-07-20)

- Increase log level of *`pykka.debug.log_thread_tracebacks()`* debugging helper from `logging.INFO` to `logging.CRITICAL`.
- Fix errors in docs examples. (PR: #29, #43)
- Fix typos in docs.
- Various project setup and development improvements.

7.4 v1.2.0 (2013-07-15)

- Enforce that multiple calls to *`pykka.Future.set()`* raises an exception. This was already the case for some implementations. The exception raised is not specified.
- Add *`pykka.Future.set_get_hook()`*.
- Add `filter()`, *`join()`*, *`map()`*, and *`reduce()`* as convenience methods using the new *`set_get_hook()`* method.
- Add support for running actors based on eventlet greenlets. See *`pykka.eventlet`* for details. Thanks to Jakub Stasiak for the implementation.

- Update documentation to reflect that the `reply_to` field on the message is private to Pykka. Actors should reply to messages simply by returning the response from *`on_receive()`*. The internal field is renamed to `pykka_reply_to` a to avoid collisions with other message fields. It is also removed from the message before the message is passed to *`on_receive()`*. Thanks to Jakub Stasiak.
- When messages are left in the actor inbox after the actor is stopped, those messages that are expecting a reply is now rejected by replying with an *`ActorDeadError`* exception. This causes other actors blocking on the returned *`Future`* without a timeout to raise the exception instead of waiting forever. Thanks to Jakub Stasiak.

  This makes the behavior of messaging an actor around the time it is stopped more consistent:

  - Messaging an already dead actor immediately raises *`ActorDeadError`*.
  - Messaging an alive actor that is stopped before it processes the message will cause the reply future to raise *`ActorDeadError`*.

  Similarly, if you ask an actor to stop multiple times, and block on the responses, all the messages will now get an reply. Previously only the first message got a reply, potentially making the application wait forever on replies to the subsequent stop messages.
- When *`ask()`* is used to asynchronously message a dead actor (e.g. `block` set to `False`), it will no longer immediately raise *`ActorDeadError`*. Instead, it will return a future and fail the future with the *`ActorDeadError`* exception. This makes the interface more consistent, as you'll have one instead of two ways the call can raise exceptions under normal conditions. If *`ask()`* is called synchronously (e.g. `block` set to `True`), the behavior is unchanged.
- A change to *`stop()`* reduces the likelyhood of a race condition when asking an actor to stop multiple times by not checking if the actor is dead before asking it to stop, but instead just go ahead and leave it to *`tell()`* to do the alive-or-dead check a single time, and as late as possible.
- Change *`is_alive()`* to check the actor's runnable flag instead of checking if the actor is registered in the actor registry.

7.5 v1.1.0 (2013-01-19)

- An exception raised in *`pykka.Actor.on_start()`* didn't stop the actor properly. Thanks to Jay Camp for finding and fixing the bug.
- Make sure exceptions in *`pykka.Actor.on_stop()`* and *`pykka.Actor.on_failure()`* is logged.
- Add *`pykka.ThreadingActor.use_daemon_thread`* flag for optionally running an actor on a daemon thread, so that it doesn't block the Python program from exiting. (Fixes: #14)
- Add *`pykka.debug.log_thread_tracebacks()`* debugging helper. (Fixes: #17)

7.6 v1.0.1 (2012-12-12)

- Name the threads of *`pykka.ThreadingActor`* after the actor class name instead of "PykkaThreadingActor-N" to ease debugging. (Fixes: #12)

7.7 v1.0.0 (2012-10-26)

- **Backwards incompatible:** Removed `pykka.VERSION` and `pykka.get_version()`, which have been deprecated since v0.14. Use *`pykka.__version__`* instead.

- **Backwards incompatible:** Removed `pykka.ActorRef.send_one_way()` and `pykka.ActorRef.send_request_reply()`, which have been deprecated since v0.14. Use *pykka.ActorRef.tell()* and *pykka.ActorRef.ask()* instead.
- **Backwards incompatible:** Actors no longer subclass `threading.Thread` or `gevent.Greenlet`. Instead they *have* a thread or greenlet that executes the actor's main loop.

  This is backwards incompatible because you no longer have access to fields/methods of the thread/greenlet that runs the actor through fields/methods on the actor itself. This was never advertised in Pykka's docs or examples, but the fields/methods have always been available.

  As a positive side effect, this fixes an issue on Python 3.x, that was introduced in Pykka 0.16, where *pykka.ThreadingActor* would accidentally override the method `threading.Thread._stop()`.
- **Backwards incompatible:** Actors that override `__init__()` *must* call the method they override. If not, the actor will no longer be properly initialized. Valid ways to call the overridden `__init__()` method include:

```
super().__init__()
# or
pykka.ThreadingActor.__init__()
# or
pykka.gevent.GeventActor.__init__()
```

- Make `pykka.Actor.__init__()` accept any arguments and keyword arguments by default. This allows you to use `super()` in `__init__()` like this:

```
super().__init__(1, 2, 3, foo='bar')
```

  Without this fix, the above use of `super()` would cause an exception because the default implementation of `__init__()` in *pykka.Actor* would not accept the arguments.
- Allow all public classes and functions to be imported directly from the *pykka* module. E.g. `from pykka.actor import ThreadingActor` can now be written as `from pykka import ThreadingActor`. The exception is *pykka.gevent*, which still needs to be imported from its own package due to its additional dependency on gevent.

7.8 v0.16 (2012-09-19)

- Let actors access themselves through a proxy. See the *pykka.ActorProxy* documentation for use cases and usage examples. (Fixes: #9)
- Give proxies direct access to the actor instances for inspecting available attributes. This access is only used for reading, and works since both threading and gevent based actors share memory with other actors. This reduces the creation cost for proxies, which is mostly visible in test suites that are starting and stopping lots of actors. For the Mopidy test suite the run time was reduced by about 33%. This change also makes self-proxying possible.
- Fix bug where *pykka.Actor.stop()* called by an actor on itself did not process the remaining messages in the inbox before the actor stopped. The behavior now matches the documentation.

7.9 v0.15 (2012-08-11)

- Change the argument of *pykka.Future.set_exception()* from an exception instance to a `exc_info` three-tuple. Passing just an exception instance to the method still works, but it is deprecated and may be unsupported in a future release.

- Due to the above change, *pykka.Future.get()* will now reraise exceptions with complete traceback from the point when the exception was first raised, and not just a traceback from when it was reraised by `get()`. (Fixes: #10)

7.10 v0.14 (2012-04-22)

- Add *pykka.__version__* to conform with **PEP 396**. This deprecates `pykka.VERSION` and `pykka.get_version()`.
- Add *pykka.ActorRef.tell()* method in favor of now deprecated `pykka.ActorRef.send_one_way()`.
- Add *pykka.ActorRef.ask()* method in favor of now deprecated `pykka.ActorRef.send_request_reply()`.
- *ThreadingFuture.set()* no longer makes a copy of the object set on the future. The setter is urged to either only pass immutable objects through futures or copy the object himself before setting it on the future. This is a less safe default, but it removes unecessary overhead in speed and memory usage for users of immutable data structures. For example, the Mopidy test suite of about 1000 tests, many which are using Pykka, is still passing after this change, but the test suite runs approximately 20% faster.

7.11 v0.13 (2011-09-24)

- 10x speedup of traversable attribute access by reusing proxies.
- 1.1x speedup of callable attribute access by reusing proxies.

7.12 v0.12.4 (2011-07-30)

- Change and document order in which *pykka.ActorRegistry.stop_all()* stops actors. The new order is the reverse of the order the actors were started in. This should make `stop_all` work for programs with simple dependency graphs in between the actors. For applications with more complex dependency graphs, the developer still needs to pay attention to the shutdown sequence. (Fixes: #8)

7.13 v0.12.3 (2011-06-25)

- If an actor that was stopped from *pykka.Actor.on_start()*, it would unregister properly, but start the receive loop and forever block on receiving incoming messages that would never arrive. This left the thread alive and isolated, ultimately blocking clean shutdown of the program. The fix ensures that the receive loop is never executed if the actor is stopped before the receive loop is started.
- Set the thread name of any *pykka.ThreadingActor* to `PykkaActorThread-N` instead of the default `Thread-N`. This eases debugging by clearly labeling actor threads in e.g. the output of `threading.enumerate()`.
- Add utility method *pykka.ActorRegistry.broadcast()* which broadcasts a message to all registered actors or to a given class of registred actors. (Fixes: #7)
- Allow multiple calls to *pykka.ActorRegistry.unregister()* with the same `pykka.actor.ActorRef` as argument without throwing a `ValueError`. (Fixes: #5)

- Make the *pykka.ActorProxy*'s reference to its *pykka.ActorRef* public as *pykka.ActorProxy.actor_ref*. The `ActorRef` instance was already exposed as a public field by the actor itself using the same name, but making it public directly on the proxy makes it possible to do e.g. `proxy.actor_ref.is_alive()` without waiting for a potentially dead actor to return an `ActorRef` instance you can use. (Fixes: #3)

7.14 v0.12.2 (2011-05-05)

- Actors are now registered in `pykka.registry.ActorRegistry` before they are started. This fixes a race condition where an actor tried to stop and unregister itself before it was registered, causing an exception in `ActorRegistry.unregister()`.

7.15 v0.12.1 (2011-04-25)

- Stop all running actors on `BaseException` instead of just `KeyboardInterrupt`, so that `sys.exit(1)` will work.

7.16 v0.12 (2011-03-30)

- First stable release, as Pykka now is used by the Mopidy project. From now on, a changelog will be maintained and we will strive for backwards compatibility.

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

O

P

R

S

T

U

V