Pykka Documentation

Release 1.2.0

Stein Magnus Jodal

February 13, 2017

Contents

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.

CHAPTER 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.

CHAPTER 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 a libevent (in 0.13) or libev (in 1.0) event loop. `GeventActor` is generally faster than `ThreadingActor`, but as of gevent 0.13 it doesn't work in processes with other threads, which limits when it can be used. With gevent 1.0, which is currently available as a release candidate, this is no longer an issue. Pykka works with both gevent 0.13 and 1.0.

- Eventlet: Each `EventletActor` is executed by a Eventlet greenlet. Pykka is tested with Eventlet 0.12.1.

Pykka has an extensive test suite, and is tested on CPython 2.6, 2.7, and 3.2+, as well as PyPy. gevent and eventlet are currently not available for CPython 3.x or PyPy.

CHAPTER 3

A basic actor

In its most basic form, a Pykka actor is a class with an `on_receive(message)` 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(Greeter, self).__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 `actor_ref`:

```
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.

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 of 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 must always be a dict, but you're mostly free to use whatever dict keys you want to.

Summarized in code:

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

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

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

future = actor_ref.ask({'msg': '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.
```

For performance reasons, Pykka **does not** clone the dict 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.

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 `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 `actor_ref.tell()` or `actor_ref.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 `actor_ref.ask()` is used and you raise an exception from within `on_receive` method it 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')
```

CHAPTER 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 buiding Mopidy.

Let's create an actor and start it:

```
import pykka

class Calculator(pykka.ThreadingActor):
    def __init__(self):
        super(Calculator, self).__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 `actor_ref.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.

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 set the `pykka_traversable` attribute to `True`:

```
import pykka

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

class Playback(object):
    pykka_traversable = True

    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 is marked as traversable.

CHAPTER 5

Examples

See the `examples/` dir in Pykka's Git repo for some runnable examples.

CHAPTER 6

What Pykka is not

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.

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

Installation

Install Pykka's dependencies:

- Python 2.6, 2.7, or 3.2+
- Optionally, Python 2.6/2.7 only:
  - gevent, if you want to use gevent based actors from `pykka.gevent`.
  - eventlet, if you want to use eventlet based actors from `pykka.eventlet`. Eventlet is known to work with PyPy 2.0 as well but Pykka is not tested with it yet.

To install Pykka you can use pip:

```
pip install pykka
```

To upgrade your Pykka installation to the latest released version:

```
pip install --upgrade pykka
```

To install the latest development snapshot:

```
pip install pykka==dev
```

CHAPTER 8

License

Pykka is licensed under the Apache License, Version 2.0.

CHAPTER 9

Project resources

- Documentation
- Source code
- Issue tracker
- CI server
- Download development snapshot

CHAPTER 10

Table of contents

10.1 Pykka API

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

10.1.1 Actors

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

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

1. subclass one of the *Actor* implementations, e.g. *GeventActor* or *ThreadingActor*,
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(MyActor, self).__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):
        ... # 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:
   (a) *actor_urn* is initialized with the assigned URN.
   (b) *actor_inbox* is initialized with a new actor inbox.
   (c) *actor_ref* is initialized with a *pykka.ActorRef* object for safely communicating with the actor.
   (d) 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.

Messages where the value of the "command" key matches "pykka_*" are reserved for internal use in Pykka.

Parameters **message** (*picklable dict*) – the message to handle

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

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.

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** (*picklable dict*) – 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 must be a picklable dict. 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** (*picklable dict*) – 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*

10.1.2 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()
```

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()
```

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):
        ...
```

An example of *ActorProxy* usage:

```
    #! /usr/bin/env python

    import pykka


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


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

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


    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.

10.1.3 Futures

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

class pykka.**Future**
A *Future* is a handle to a value which are 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()*.

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 picklable 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.

Changed in version 0.15: Previously, *`set_exception()`* accepted an exception instance as its only argument. This still works, but it is deprecated and will be removed in a future release.

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.

If the future's result is a single value, it is simply passed to the function. If the future's result is an iterable, the function is applied to each item in the iterable.

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 + 10)
>>> f.set([30, 300, 3000])
>>> g.get()
[40, 310, 3010]
```

New in version 1.2.

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([], 5)
>>> g.get()
5
```

New in version 1.2.

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.

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

10.1.4 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** (*picklable dict*) – 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

10.1.5 Gevent support

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`

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, but as of gevent 0.13.x it does not work in combination with other threads.

10.1.6 Eventlet support

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*.

class pykka.eventlet.**EventletActor**(**args*, ***kwargs*)
EventletActor implements *pykka.Actor* using the eventlet library.

This implementation uses eventlet green threads.

10.1.7 Logging

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

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 records from the library will be exposed to the application's users. In other words, if you want to see the log records 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 statements out of Pykka:

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

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)
```

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

10.1.8 Debug helpers

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

Logs at `INFO` 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.INFO` 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.

For a complete example of how to use this, see `examples/deadlock_debugging.py` in Pykka's source code.

New in version 1.1.

10.2 Changes

10.2.1 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 raises 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 registrered in the actor registry.

10.2.2 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)

10.2.3 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)

10.2.4 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(MyActorSubclass, self).__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(MyActorSubclass, self).__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.

10.2.5 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.

10.2.6 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)

10.2.7 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.

10.2.8 v0.13 (2011-09-24)

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

10.2.9 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)

10.2.10 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)

10.2.11 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()`.

10.2.12 v0.12.1 (2011-04-25)

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

10.2.13 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 compatability.

d

e

g

p

Symbols

A

B

E

F

G

I

J

L

M

O

P

R

S

T

U