pyactor Documentation

Release 2.0.1

URV

Apr 14, 2020

Contents

CHAPTER 1

The minimalistic python actor middleware

PyActor is a python actor middleware for an object oriented architecture constructed with the idea of getting two remote actors to quickly communicate in a very simple, lightweight and minimalistic way.

It supports two versions:

- Threading
- Gevent green threads

It also includes communication between machines using XMLRPC and a second version that uses RabbitMQ message system in a transparent way.

For install instructions, see *Installation*.

Visit the repository at GitHub.

See the tuturial for a complete review on the features of PyActor. And complete it with the *Remote Tutorial*.

CHAPTER 2

Contents:

2.1 Tutorial

A quick guide on how to use the PyActor library through examples.

2.1.1 Installation

This library allows the creation and management of actors in a distributed system using Python. It follows the classic actor model and tries to be a simple way to get two remote actors to quickly communicate.

To install the library, use:

```
python setup.py install
```

You can check that works with the examples explained in this page, that you can find in the ./examples directory of this project.

The library requires Gevent.

It is also available at PYPI, so the most easy way of installing PyActor is by:

```
pip install pyactor
```

Then you can check the examples from the repository.

2.1.2 Global indications

This library is implemented using two types of concurrence: threads and green threads (Gevent). To define which one you want, always use the function *set_context()* at the beginning of your script. The default value uses threads but you can specify the mode with one of the following strings:

- `'thread'`
- `'green_thread'`

Then, first of all, a *Host* is needed in order to create some actors. To create a host, use the function *create_host()* which returns a proxy (*Proxy*) to the instance of a *Host*. You should never work with the instance itself, but always with proxies to send messages to actors. When you have the proxy, use it to spawn actors by giving the class type of the actor to create and one string that will identify it in the host. The *spawn()* method will return the proxy that manages that actor. See example:

```
h = create_host()
actor1 = h.spawn('id1', MyClass)
```

The class of an actor must have defined its methods in the _tell and _ask sets so they can be called through the proxy. In the _tell set will be named those methods meant to be asynchronous and in the _ask set, the synchronous ones. In this example we have a class MyClass with a sync method *ask_me()* and an async method *tell_me()*:

```
class MyClass:
    _tell = {'tell_me'}
    _ask = {'ask_me'}
    def tell_me(self, msg):
        print(msg)
    def ask_me(self):
        return "hello back"
```

As you can see, the async method receives a message and simply prints it while the sync method returns a result. More detailed examples can be found in the 'pyactor/examples' directory of the project. They are also explained below as a tutorial for this library.

2.1.3 Sample 1 - Basic

This example shows and tests the most basic elements of PyActor. It creates a *Host* and adds an actor to it. Then, queries an async method of this actor. This is the full code of this sample, which you can find and test in `pyactor\examples\sample1.py`:

```
"""
Basic host creation sample.
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo'}

    def echo(self, msg):
        print(msg)


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    e1.echo("hello there !!")

    sleep(1)
    shutdown()
```

The example is similar to the one shown above in *Global indications*, but here we'll explain it more carefully.

In this case, we need to import the *create_host()* function from the project in order to use it. We also import the *sleep* function, to give time to the actor to work, and the setting function for the type, *set_context()*. Finally, we

also need the *shutdown()* function to stop and clean the host before finishing.

The actor to create in this example will be an `Echo`. This class only has one method which prints the message *msg*, given by parameter. As you can see, the classes destined to be actors must have the attributes `_tell={...}` and `_ask={...}` that include the names of the methods that can be remotely invoked in an asynchronous or synchronous way, respectively. In this sample we have the echo method, which is async, as no response from it is needed.

Note: In this sample we do not have synchronous methods, so it is not necessary to declare the _ask set. However, it could also be declared as an empty ser `_ask = set()`.

The first thing to do is define which model are we going to use. For now, we are using the classic threads, so we'll call the function without parameters to use the default solution.

```
set_context()
```

To begin the execution we'll need a *Host* to contain the actors. For that, we create a new variable by using the function we imported before.

```
h = create_host()
```

Now we have a *Host* in the 'h' variable. Actually, as Host objects are also actors, this call returns a `Proxy` that will manage that actor. It can create actors attached to itself. To do that, we use the *spawn()* method. The first parameter is a string with the ID of the actor that will identify it among the host, so no repeated values are allowed. The second is the class the actor will be instance of. In this case we create an actor which will be an `Echo` and with the id 'echo1':

```
e1 = h.spawn('echo1', Echo)
```

'e1' will now represent that actor (actually, it's a `Proxy` that manages it).

As we have the actor, we can invoke its methods as we would do normally since the proxy will redirect the queries to the actual placement of it. If we didn't have specified the methods in the statements appointed before (_tell and _ask), we wouldn't be able to do this now, giving a 'no such attribute error'. The execution should work properly and print on screen:

```
hello there !!
```

Then, the sleep gives time to the actor for doing the work and finally, we close the host, which will stop all its actors. This function (*shutdown()*) should be always called at the end to do a clean exit:

```
shutdown()
```

Note: As the host is an actor itself, it has sync and async methods and can receive remote queries if we use its proxy.

Note: As said, the host is also a living actor so it could receive queries remotely in the future. This means you can send its reference to another host, which allows to spawn remotely (remote spawns require a bit more info, see the remote tutorial).

Note: Now you can try and see how it works with green threads by just specifying 'green_thread' in the setting function. `set_context('green_thread')`

2.1.4 Sample 2 - Sync

This example extends the content of the previous one by including sync requests. It still creates a *Host* and adds an actor to it. This is the full code of this sample, which you can find and test in pyactor\examples\sample2.py:

```
"""
Sync/async queries sample.
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo', 'bye'}
    _ask = {'say_something'}

    def echo(self, msg):
        print(msg)

    def bye(self):
        print("bye")

    def say_something(self):
        return "something"


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    e1.echo("hello there !!")
    e1.bye()

    print(e1.say_something())

    sleep(1)
    shutdown()
```

Now Echo has two new methods, bye() and say_something(). The first one is async like the previous echo(), but the other one is synchronous.

The invocation of ask methods is simply the same you would do normally.

The correct output for this sample is the following:

```
hello there !!
bye
something
```

2.1.5 Sample 3 - Timeout

This example tests the raising of timeouts. This is the full code of this sample, which you can find and test in pyactor\examples\sample3.py:

```
"""
Timeout sample.
"""
```

(continues on next page)

(continued from previous page)

```
from pyactor.context import set_context, create_host, sleep, shutdown
from pyactor.exceptions import PyActorTimeoutError


class Echo(object):
    _tell = {'echo', 'bye'}
    _ask = {'say_something'}

    def echo(self, msg):
        print(msg)

    def bye(self):
        print("bye")

    def say_something(self):
        sleep(2)
        return "something"


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    e1.echo("hello there !!")
    e1.bye()

    try:
        x = e1.say_something(timeout=1)
    except PyActorTimeoutError:
        print("timeout caught")
    sleep(1)
    shutdown()
```

Now we have the same `Echo` class but in the sync method we added a sleep of 2 seconds. Also, we surrounded the method call by a try structure catching a *`PyActorTimeoutError`* exception from pyactor.exceptions. Since we are giving an expire time of 1 second to the invocation, the timeout will be reached and the exception raised.

You can set a timeout for the query of your choice. For that, add the keyword parameter *timeout=X* in the call, in seconds.

```
x = e1.say_something(timeout=3)
```

The default timeout is 10 seconds. To wait indefinitely, just set it to *None*, but that is not recommended.

The correct output for this sample is the following:

```
hello there !!
bye
timeout caught
```

2.1.6 Sample 4 - Lookup

This example shows the usage of the lookup methods applied to a host. This is the full code of this sample, which you can find and test in `pyactor\examples\sample4.py`:

```
"""
Lookup sample.
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo', 'bye'}
    _ask = {'say_something'}

    def echo(self, msg):
        print(msg)

    def bye(self):
        print("bye")

    def say_something(self):
        return "something"


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)

    e = h.lookup('echo1')
    print(e.say_something())

    ee = h.lookup_url("local://local:6666/echo1", Echo)
    print(ee.say_something())

    sleep(1)
    shutdown()
```

We have two ways to get the reference of one already existing actor of a host. If it is local, of the same host, it is fine to use the method *lookup()* giving by parameter only the id of the actor you wish:

```
e = h.lookup('echo1')
```

If you are working remotely, you could need *lookup_url()* to get the reference. In this example, it is used also to get a local reference giving the standard local URL at which the host is initialized by default:

```
ee = h.lookup_url('local://local:6666/echo1')
```

Note: Please follow the remote tutorial to get a better overview of the programming with remote hosts. This tutorial focuses on local hosts.

2.1.7 Sample 5 - References to actors

This example tests the sending of proxy references by parameter using the definition of the _ref set. This is the full code of this sample, which you can find and test in `pyactor\examples\sample5.py`:

```
"""
Proxy references by parameter sample.
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo', 'echo2', 'echo3'}
    _ref = {'echo', 'echo2', 'echo3'}

    def echo(self, msg, sender):
        print(f"{msg} from: {sender.get_name()}")

    def echo2(self, msg, senders):
        for sender in senders:
            print(f"{msg} from: {sender.get_name()}")

    def echo3(self, msg, senders):
        for sender in senders.values():
            print(f"{msg} from: {sender.get_name()}")


class Bot(object):
    _ask = {'get_name'}

    def get_name(self):
        return self.id


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    bot = h.spawn('bot1', Bot)
    bot2 = h.spawn('bot2', Bot)
    sleep(1)
    e1.echo("HI!", bot)
    e1.echo2("hello there!", [bot2])
    e1.echo3("hello there!!", {'bot1': bot, 'bot2': bot2})

    sleep(1)
    shutdown()
```

If you pass references to actors (proxies) by parameter in actors methods, would mean they are sharing the same instance of a proxy. This could cause various concurrency problems, so we might want different proxies in different spots. To achieve that, you have to indicate that a method *receives or returns* a proxy by adding it to the class' _ref set (it still must be in _ask or _tell).

With this indication, PyActor will search for proxies in the parameters and make a new proxy for the actor in the context that the method will be executed.

In the example, Echo has methods that receive a proxy, in this methods you can see examples of passing proxies even inside lists or dictionaries. For that to work correctly on any system, Echo needs to define its methods as they have this functionality. This is why all three methods are in the _ref set

```
_ref = {'echo', 'echo2', 'echo3'}
```

Although the proxies are different, you may yet compare them directly so when using `p1 == p2` on two proxies, the

comparison will be done on the actors that they represent and not on the proxy instance itself. See the basic examples on `proxies_test.py`.

2.1.8 Sample 6 - self.id, proxy and host

This example tests the self references to an actor's id and proxy. This is the full code of this sample, which you can find and test in `pyactor\examples\sample6.py`:

```
"""
Self references sample. Actor id/proxy. + serve_forever
"""
from pyactor.context import set_context, create_host, sleep, serve_forever


class Echo(object):
    _tell = {'echo'}
    _ref = {'echo'}

    def echo(self, msg, sender):
        print(f"{msg} from: {sender.get_name()} at {sender.get_net()}")
        # print(sender.get_id(), sender.get_url())


class Bot(object):
    _tell = {'set_echo', 'say_hi'}
    _ask = {'get_name', 'get_net'}

    def __init__(self):
        self.greetings = ["hello", "hi", "hey", "what's up?"]

    def set_echo(self):
        self.echo = self.host.lookup('echo1')

    def get_name(self):
        return self.id

    def get_net(self):
        return self.url

    def say_hi(self):
        for salute in self.greetings:
            self.echo.echo(salute, self.proxy)


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    bot = h.spawn('bot1', Bot)
    bot.set_echo()
    bot.say_hi()

    sleep(1)
    serve_forever()
```

This sample demonstrates how to get references to an actor from the actor itself. With `self.id` we obtain the string that identifies the actor in the host it is located, `self.url` contains its network location. Then, with `self.proxy`

you can get a reference to a proxy managing the actor so you can give it to another function, class or module in a safe and easy way.

Note: Remember to put methods that receive or return proxies in the _ref set.

It is also possible to use `self.host`, which will give a proxy to the host in which the actor is, so you can *lookup()* other actors from there, among other possibilities.

In the example, we use these three calls to send various salutations from a `Bot` to an `Echo` giving by parameter also a proxy from the Bot so the Echo can call one of the Bot's methods to get its id. Also, the `set_echo()` method, in this case, does not receive the Echo by parameter. It uses the inside reference it already has to call a *lookup()* to the host and get the wanted reference.

Also notice that every proxy has the methods `get_id` and `get_url` already defined, so you can get the actor's information directly from the proxy. This means we could use `sender.get_id()` instead of `sender.get_name()`; and `sender.get_url()` instead of `sender.get_net()` on the echo method.

The correct output for this sample is the following:

```
hello from: bot1
hi from: bot1
hey from: bot1
what`s up? from: bot1
Press Ctrl+C to kill the execution
```

In this sample, we also see the usage of the *serve_forever()* function which is very useful in remote communication in order to keep a host alive as another one sends queries to its actors. The usage is very simple, instead of shutting the host down at the end, we call:

```
serve_forever()
```

This will maintain the host alive in lower process consumption until the user presses `Ctrl+C` allowing other hosts to lookup and call methods from actors in this host.

2.1.9 Sample 7 - References extended

This example extends sample 5. This is the full code of this sample, which you can find and test in `pyactor\examples\sample7.py`:

```
"""
Proxy references by parameter sample.
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo', 'echo2', 'echo3'}
    _ref = {'echo', 'echo2', 'echo3'}

    def echo(self, msg, sender):
        print(f"{msg} from: {sender.get_name()}")

    def echo2(self, msg, senders):
        for sender in senders:
            print(f"{msg} from: {sender.get_name()}")
```

(continues on next page)

(continued from previous page)

```
    def echo3(self, msg, senders):
        for sender in senders.values():
            print(f"{msg} from: {sender.get_name()}")


class Bot(object):
    _tell = {'set_echo', 'say_hi'}
    _ask = {'get_name'}
    _ref = {'set_echo'}

    def __init__(self):
        self.greetings = ["hello", "hi", "hey", "what's up?"]

    def set_echo(self, echo):
        self.echo = echo

    def get_name(self):
        return self.id

    def say_hi(self):
        for salute in self.greetings:
            self.echo.echo(salute, self.proxy)


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    bot = h.spawn('bot1', Bot)
    bot2 = h.spawn('bot2', Bot)
    bot.set_echo(e1)     # Passing a proxy to a method marked as _ref
    sleep(1)             # Give time to host to lookup the first one
    bot2.set_echo(e1)
    bot.say_hi()
    sleep(1)
    e1.echo2("hello there!", [bot2])
    e1.echo3("hello there!!", {'bot1': bot, 'bot2': bot2})

    sleep(1)
    shutdown()
```

To remark the importance of using the _ref set, we extend here sample 5 with more examples of passing proxies combined with the self references we saw in sample 6.

Bot has a method `set_echo` that gets the echo it will use by parameter. As this echo has to be a proxy, Bot includes the next definition:

```
_ref = {'set_echo'}
```

So then, at the main code, we can make this call without any concurrency problems, as the proxies are not shared:

```
bot.set_echo(e1)
```

As already seen in sample 5, Echo has methods that receive a proxy. Including examples of passing proxies even inside lists or dictionaries.

2.1.10 Sample 8 - Futures

This example tests more deeply the features of futures. This is the full code of this sample, which you can find and test in `pyactor\examples\sample8.py`:

```
"""
Futures Sample.
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo'}
    _ask = {'say_something', 'raise_something'}

    def echo(self, msg):
        print(msg)

    def say_something(self):
        return "something"

    def raise_something(self):
        raise Exception("raising something")


if __name__ == '__main__':
    set_context()
    # set_context('green_thread')
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    e1.echo("hello there !!")

    # ask = e1.raise_something(future=True)
    ask = e1.say_something(future=True)
    print(f"Future: {ask}")
    sleep(0.1)
    if ask.done():
        print(f"Exception: {ask.exception()}")
        try:
            print(f"Result: {ask.result(1)}")
        except Exception as e:
            print(e)

    sleep(1)
    shutdown()
```

The example is like Sample 3, but here we use the futures approach.

We do this by adding the parameter *future=True* to the call. This will make the query return a *Future* instance instead of the result. That means that the execution of the query may have not been completed yet. To get the result from a Future, use the method *result()* as you can see in the try section.

Also shows the usage of the consulting methods of futures: *done()*, and *exception()*.

Change between this lines:

```
ask = e1.raise_something(future=True)
ask = e1.say_something(future=True)
```

to check the raising of exceptions.

Finally, note that the only argument for *result()* (also for *exception()*) is the timeout: the time, in seconds, to wait for a result before raising an error.

2.1.11 Sample 9 - Callback

This example tries the functionality of the callback element of the synchronous queries. This is the full code of this sample, which you can find and test in `pyactor\examples\sample9.py`:

```
"""
Callback sample.
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo', 'bye'}
    _ask = {'say_something'}

    def echo(self, msg):
        print(msg)

    def bye(self):
        print("bye")

    def say_something(self):
        sleep(1)
        return "something"


class Bot(object):
    _tell = {'set_echo', 'ping', 'pong'}
    _ref = {'set_echo'}

    def set_echo(self, echo):
        self.echo = echo

    def ping(self):
        future = self.echo.say_something(future=True)
        future.add_callback('pong')
        future.add_callback('pong')
        print("pinging...")

    def pong(self, future):
        msg = future.result()
        print("callback", msg)


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    bot = h.spawn('bot', Bot)
    bot.set_echo(e1)
    bot.ping()

```

(continues on next page)

(continued from previous page)

```
48      sleep(2)
49      shutdown()
```

This time we keep having the same initialization as before, but now there is a new class. `Bot` has three async methods that will allow to prove the callback functionality. `set_echo()` registers an `Echo` to the Bot so it can call it. *ping* creates the query for the `say_something()` method and sets the callback for this to his other method `pong()`. This second will receive the result of the execution of the `say_something()` method.

Remember, `set_echo()` needs to be listed in the `Bot` class' *_ref* set.

In order to add a callback, the sync call must be defined as a Future. We do this by adding the parameter *future=True* to the call.

Then, use the Future method *`add_callback()`* which takes by parameter the name of the method to callback, which is one from the actor that calls it. You can add various callbacks to one future, and they will be called in order when the work is finished. Also, if you add a callback to a finished future, it will be directly invoked.

See *Sample 8 - Futures* for a more complex sample on Futures.

Note: *`add_callback()`* needs to be called from inside an actor, specifying a method of that same actor.

Note: The method treated as a callback must have one unique parameter, which is the future. Inside the method you can use *`result()`* to get the result of the call (exceptions can be raised) or *`exception()`* to get the instance of a possible raised exception. You can also check the state of the future with one of its methods: *`done()`* or *`running()`*.

The correct output for this sample is the following:

```
pinging...
callback something
callback something
```

2.1.12 Sample 10 - Parallel

This example tests the creation and execution of actors with parallel methods. This is the full code of this sample, which you can find and test in `pyactor\examples\sample10.py`:

```
1   """
2   Parallel methods sample.
3   """
4   from pyactor.context import set_context, create_host, sleep, shutdown
5   from pyactor.exceptions import PyActorTimeoutError
6
7
8   class File(object):
9       _ask = {'download'}
10
11      def download(self, filename):
12          print(f"downloading {filename}")
13          sleep(5)
14          return True
15
16
```

(continues on next page)

(continued from previous page)

```
class Web(object):
    _ask = {'list_files', 'get_file'}
    _tell = {'remote_server'}
    _parallel = {'list_files', 'get_file', 'remote_server'}
    # Comment the line above to check the raise of timeouts if parallels
    # are not used.
    _ref = {'remote_server'}

    def __init__(self):
        self.files = ["a1.txt", "a2.txt", "a3.txt", "a4.zip"]

    def remote_server(self, file_server):
        self.server = file_server

    def list_files(self):
        return self.files

    def get_file(self, filename):
        return self.server.download(filename, timeout=6)


class Workload(object):
    _tell = {'launch', 'download', 'remote_server'}
    _ref = {'remote_server'}

    def launch(self):
        for i in range(10):
            try:
                print(self.server.list_files(timeout=2))
            except PyActorTimeoutError as e:
                print(i, e)

    def remote_server(self, web_server):
        self.server = web_server

    def download(self):
        self.server.get_file('a1.txt', timeout=10)
        print("download finished")


if __name__ == '__main__':
    set_context('green_thread')
    # set_context()

    host = create_host()

    f1 = host.spawn('file1', File)
    web = host.spawn('web1', Web)
    sleep(1)
    web.remote_server(f1)
    load = host.spawn('wl1', Workload)
    load.remote_server(web)
    load2 = host.spawn('wl2', Workload)
    load2.remote_server(web)

    load.launch()
    load2.download()
```

(continues on next page)

(continued from previous page)

```
74
75      sleep(7)
76      shutdown()
```

Parallels are a way of letting one actor process many queries at a time. This will allow the actor to keep receiving calls when another call has been blocked with another job (an I/O call or a synchronous call to another actor).

To make one method execute parallel, you need to specify it in the class attribute _parallel, which is a set. The method must also be in one of the sets _tell or _ask. The methods with this tag will be executed in new threads so their execution do not interfere with receiving other queries. That is, the actor can attend other queries while executing the parallel method.

As you could think, executing methods of the same actor at the same time can compromise the integrity of data. PyActor ensures that only one thread is executing on an actor at the same time, allowing other threads to execute when the one executing is blocked with some call. This prevents two threads from accessing the same data at a time, but is up to the programmer to prevent the data to change during the execution of a method if that is not intended, as a method could modify a property of the actor while a parallel, that operates with that data, is blocked, leading to an inconsistency.

In this example we have three classes: File, Web and Workload. File represents a server that serves the download of files. Simulates the work with a sleep. Web represents a web server which contains a list of files. It must have a file server that provides the files and can list its files (list_files) and return one of them (get_file). Workload is the class that will do the work. It asks the web to list its files ten times, or requests to download one of the files.

The execution is simple, we create one file server, one web server and attach the file server to the web:

```
web.remote_server(f1)
```

Then let's do the work. Create two Workload instances and pass to them the web server we created:

```
load = host.spawn('wl1', Workload)
load.remote_server(web)
load2 = host.spawn('wl2', Workload)
load2.remote_server(web)
```

The first worker will make the ten queries to list_files, while the second one will download a file:

```
load.launch()
load2.download()
```

As the method get_file is marked as parallel, its execution will be done in another thread, so when the method blocks downloading (in the sleep), it will free the actor so it can keep serving answers to the first load.

If we do not use parallels in this example (which you can try by commenting the right line as indicated) some of the calls to the list_files method will raise TimeoutError as that actor's thread is blocked with the download.

Note: *sample10b* combines this example with the use of Futures.

Note: You can test another parallel example with *parall.py*. That might result simpler to follow.

2.1.13 Sample 11 - Intervals

This example tests the usage of intervals that allow an actor to periodically do an action. This is the full code of this sample, which you can find and test in `pyactor\examples\sample11.py`:

```
"""
Intervals sample
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, sleep, shutdown, \
    interval, later


class Registry(object):
    _tell = {'hello', 'init_start', 'stop_interval'}
    # _ref = {'hello'}

    def init_start(self):
        self.interval1 = interval(self.host, 1, self.proxy, "hello", "you", "too")
        later(5, self.proxy, "stop_interval")

    def stop_interval(self):
        print("stopping interval")
        self.interval1.set()

    def hello(self, msg, m2):
        print(f"{self.id} Hello {msg} {m2}")


if __name__ == '__main__':
    N = 2   # 10000

    set_context()
    host = create_host()
    registry = list()
    for i in range(0, N):
        registry.append(host.spawn(str(i), Registry))

    for i in range(0, N):
        registry[i].init_start()

    sleep(8)
    shutdown()
```

To generate intervals, we use the functions `context.interval()` and `context.later()` that can be imported if needed. The class (actor) will call the first one giving firstly the proxy of the host that will manage the interval, accessible from within the actor by *self.host*; next, the interval time and the proxy to the actor to which make the periodic call (that can be itself with *self.proxy* or another actor) as well as the name of the method in that actor that will be called. The method to be executed must be a tell method (with ref or without it), otherwise, it will raise and exception.

This function returns an interval instance that we have to keep in order to stop it later by calling *.set()*.

In this example we use `context.later()` to set a timer that will stop the interval after a certain time. This method works similar to the other. You specify by parameter the actor and the method to be executed after that time, and only accepts methods of the tell type.

If the method requires arguments, those can be passed in the same call. In the example, hello needs one argument and it is passed as:

```
self.host.interval(1, self.proxy, "hello", "you")
```

If the method needed two of them, it would be like follows:

```
self.host.interval(1, self.proxy, "hello", "you", "too")
```

2.1.14 Sample 1b - Stopping an Actor (Advanced)

This example is like the first one, but extended with a new functionality for the hosts. This shows how to stop an actor and delete all its references from one host. This is the full code of this sample, which you can find and test in `pyactor\examples\sample1b.py`:

```
"""
Stopping an actor.
"""
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo'}

    def echo(self, msg):
        print(msg)


if __name__ == '__main__':
    set_context()
    h = create_host()
    e1 = h.spawn('echo1', Echo)
    e1.echo("hello there !!")

    sleep(1)
    h.stop_actor('echo1')

    e1 = h.spawn('echo1', Echo)
    e1.echo("hello there !!")

    sleep(1)
    shutdown()
```

You can always delete an actor by calling the method *stop_actor()* of its host. This function will stop the thread of that actor and all its references from the host. This means the actor cannot be looked up anymore, it will not receive any more work and you can create a new actor with its same id.

Note: Parallel queries already submitted will end as usual.

Note: Intervals involving that actor's methods might result in errors.

2.2 Remote Tutorial

This page explains hot to use PyActor for remote communications between machines.

2.2.1 Sample 1 - Basic communication

This example shows the basis on setting a remote communication and sending tell requests. This is the full code of this sample, which you can find and test in `pyactor\examples\Remote\s1_server.py`:

```
"""
Basic remote example sending tell messages. SERVER
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, serve_forever


class Echo(object):
    _tell = {'echo'}

    def echo(self, msg):
        print(msg)


if __name__ == '__main__':
    set_context()
    host = create_host("http://127.0.0.1:1277/")

    e1 = host.spawn('echo1', Echo)
    serve_forever()
```

And `pyactor\examples\Remote\s1_client.py`:

```
"""
Basic remote example sending tell messages. CLIENT
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, shutdown


if __name__ == '__main__':
    set_context()
    host = create_host("http://127.0.0.1:1679")

    e1 = host.lookup_url("http://127.0.0.1:1277/echo1", 'Echo', 's1_server')

    e1.echo("Hi there!")     # TELL message
    e1.echo("See ya!")

    shutdown()
```

To create a host able to communicate with other machines, simply use as its URL one with an http scheme, as in the example. Using the http scheme will create a dispatcher on that host that will manage the queries through xml.

So, the server spawns an actor at `127.0.0.1:1277` and the client is able to look for that actor just giving that IP:port and path. If the client does not have the Class it is looking for, it must provide the module and the name of that class when calling the lookup method as shown.

Then, the calls are used as usual.

In `s1_clientb.py` we have the same code but the calls are repeated 1000 times.

2.2.2 Sample 2 - Basic communication 2

This example extends the first by adding ask requests. This is the full code of this sample, which you can find and test in `pyactor\examples\Remote\s2_server.py`:

```
"""
Basic remote example sending ask messages. SERVER
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, serve_forever


class Echo(object):
    _tell = {'echo'}
    _ask = {'get_msgs'}

    def __init__(self):
        self.msgs = []

    def echo(self, msg):
        print(msg)
        self.msgs.append(msg)

    def get_msgs(self):
        return self.msgs


if __name__ == '__main__':
    set_context()
    host = create_host("http://127.0.0.1:1277/")

    e1 = host.spawn('echo1', Echo)
    serve_forever()
```

And `pyactor\examples\Remote\s2_client.py`:

```
"""
Basic remote example sending ask messages. CLIENT
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, shutdown


if __name__ == '__main__':
    set_context()
    host = create_host("http://127.0.0.1:1679")

    e1 = host.lookup_url("http://127.0.0.1:1277/echo1", 'Echo', 's2_server')

    e1.echo('Hi there!')     # TELL message
    e1.echo('See ya!')

    print(e1.get_msgs())
```

(continues on next page)

(continued from previous page)

```
18
19      shutdown()
```

This sample is like the previous one, but it includes examples of ask methods. As the tell methods, they are used as normally, like in the local examples.

2.2.3 Sample 3 - Remote spawning

This example shows how to spawn an actor in another host. This is the full code of this sample, which you can find and test in `pyactor\examples\Remote\s3_host.py`:

```
1  """
2  Remote example spawning on a remote server. SERVER
3  @author: Daniel Barcelona Pons
4  """
5  from pyactor.context import set_context, create_host, serve_forever
6
7
8  if __name__ == '__main__':
9      set_context()
10     host = create_host("http://127.0.0.1:1277/")
11
12     print("host listening at port 1277")
13
14     serve_forever()
```

And `pyactor\examples\Remote\s3_client.py`:

```
1  """
2  Remote example spawning on a remote server. CLIENT
3  @author: Daniel Barcelona Pons
4  """
5  from pyactor.context import set_context, create_host, Host, sleep, shutdown
6  from pyactor.exceptions import PyActorTimeoutError
7
8
9  class Server(object):
10     _ask = {'add', 'wait_a_lot'}
11     _tell = {'substract'}
12
13     def add(self, x, y):
14         return x + y
15
16     def substract(self, x, y):
17         print("subtract", x - y)
18
19     def wait_a_lot(self):
20         sleep(2)
21         return "ok"
22
23
24 if __name__ == '__main__':
25     set_context()
26     host = create_host("http://127.0.0.1:1679")
27
```

(continues on next page)

(continued from previous page)

```
    remote_host = host.lookup_url("http://127.0.0.1:1277/", Host)
    print(remote_host)
    server = remote_host.spawn('server', 's3_client/Server')
    z = server.add(6, 7)
    print(z)
    server.subtract(6, 5)
    t = server.add(8, 7)
    print(t)

    try:
        print(server.wait_a_lot(timeout=1))
    except PyActorTimeoutError as e:
        print(e)

    sleep(3)
    shutdown()
```

In this case the server part only creates its host and makes it serve forever (`serve_forever()`). The client is the one that uses *`lookup_url()`* to get the server reference and spawn an actor in it. Then, sends the work to the actor. To spawn the actor, as the class of it is defined in the client module, the method uses a string to define where is the Class so the server can import it. This string uses the form `module/class_name`:

```
server = remote_host.spawn('server', 's3_client/Server')
```

2.2.4 Sample 4 - Registry example

Here we have a basic example of a registry where some servers can bind to so the clients are able to see all the servers available and connect to one. This is the full code of this sample, which you can find and test in `pyactor\examples\Remote\s4_registry.py`:

```
"""
Remote example with a registry. SERVER
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, serve_forever


class NotFound(Exception):
    pass


class Registry(object):
    _ask = {'get_all', 'bind', 'lookup', 'unbind'}
    _ref = {'get_all', 'bind', 'lookup'}

    def __init__(self):
        self.actors = {}

    def bind(self, name, actor):
        print("server registred", name)
        self.actors[name] = actor

    def unbind(self, name):
        if name in self.actors.keys():
```

(continues on next page)

(continued from previous page)

```
            del self.actors[name]
        else:
            raise NotFound()

    def lookup(self, name):
        if name in self.actors:
            return self.actors[name]
        else:
            return None

    def get_all(self):
        return self.actors.values()


if __name__ == '__main__':
    set_context()
    host = create_host("http://127.0.0.1:6000/")

    registry = host.spawn('regis', Registry)

    print("host listening at port 6000")

    serve_forever()
```

And `pyactor\examples\Remote\s4_client.py`:

```
"""
Remote example with registry. CLIENT
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, serve_forever


if __name__ == '__main__':
    set_context()
    host = create_host("http://127.0.0.1:6001")

    registry = host.lookup_url("http://127.0.0.1:6000/regis", 'Registry',
                               's4_registry')

    registry.bind('host1', host)

    serve_forever()
```

And `pyactor\examples\Remote\s4_clientb.py`:

```
"""
Remote example with registry. CLIENT 2
@author: Daniel Barcelona Pons
"""
from pyactor.context import set_context, create_host, sleep, shutdown

from s4_registry import NotFound


class Server(object):
    _ask = {'add', 'wait_a_lot'}
```

(continues on next page)

(continued from previous page)

```
    _tell = {'subtract'}

    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        print("subtract", x - y)

    def wait_a_lot(self):
        sleep(2)
        return "ok"


if __name__ == '__main__':
    set_context()
    host = create_host("http://127.0.0.1:6002")

    registry = host.lookup_url("http://127.0.0.1:6000/regis", 'Registry',
                               's4_registry')
    remote_host = registry.lookup('host1')
    if remote_host is not None:
        if not remote_host.has_actor('server'):
            server = remote_host.spawn('server', 's4_clientb/Server')
        else:
            server = remote_host.lookup('server')
        z = server.add(6, 7)
        print(z)
        server.subtract(6, 5)
        t = server.add(8, 7)
        print(t)

    try:
        registry.unbind('None')
    except NotFound:
        print("Cannot unbind this object: is not in the registry.")

    shutdown()
```

In this example we have a registry where Servers can be bound. The registry module starts an actor which is the registry itself to which servers can be bound and clients look for servers. The first client binds its host to the registry and waits. The second one uses the registry to find the first's host and spawn a server on it. Then, send work to that server.

In order to execute the second client repeatedly without having to restart all the processes, before spawning the server remotely, it checks if the first client has already the server by using the method `has_actor` on the remote_host.

2.2.5 Sample 5 - Multiple Hosts

This example tests the creation of multiple host at the same time on one unique execution. This is the full code of this sample, which you can find and test in `pyactor\examples\Remote\sample5.py`:

```
"""
Multiple hosts. Remote required since v0.9.
@author: Daniel Barcelona Pons
"""
```

(continues on next page)

(continued from previous page)

```
from pyactor.context import set_context, create_host, sleep, shutdown


class Echo(object):
    _tell = {'echo'}
    _ref = {'echo'}

    def echo(self, msg, pref=None):
        print(msg, pref)


if __name__ == '__main__':
    set_context()
    h = create_host("http://127.0.0.1:6666/host")
    e1 = h.spawn('echo1', Echo)
    e1.echo("hello there !!", e1)

    h2 = create_host("http://127.0.0.1:7777/host")
    e2 = h2.spawn('echo1', Echo)
    e2.echo("hello 2", e1)

    sleep(1)

    e1.echo("hello 3", e2)

    sleep(1)
    shutdown()
    # or, to only stop one of them:
    # shutdown("http://127.0.0.1:7777/host")
```

The first thing to make clear is that you should never need to create more than one host locally, since they are meant for remote communication. This is for testing purposes.

To create more hosts, you only need to call again the function *create_host()*. But you will need to specify different locations for each host, since those are their identifiers. In the example we create two hosts in the same location, but attending different ports:

```
h = create_host("http://127.0.0.1:6666/host")
h2 = create_host("http://127.0.0.1:7777/host")
```

Note: Remember that the default address for a host is `local://local:6666/host`

Note: To communicate two hosts, both of them must have a remote dispatcher, so they must have one of the schemes required.

Now, each host will manage its own actors and threads, so they will need to communicate through TCP connections.

One thing important to know about this is that only one host can be used to manage the main execution of your program, so there always will be a main host and the other ones will be created as secondary hosts.

This main host will be automatically assigned to the first one created. If that one is closed and there still are other hosts operative, the oldest of them will assume the role of main host.

2.2.6 Using RabbitMQ

Unmaintained Only works on a single machine with multiple hosts and needs the rabbit server running locally.

This library also supports the usage of communication through RabbitMQ queues. To use this approach, simply define the hosts with an URL with the scheme *amqp* instead of *http*. This will create a dispatcher for that host that works with RabbitMQ, and all its actors will work at that scheme.

You can see an example with `pyactor\examples\Remote\s1_clientrbb.py`:

```
"""
Basic remote example sending tell messages. CLIENT
@author: Daniel Barcelona Pons
"""
from pyactor.context import \
    set_context, create_host, set_rabbit_credentials, shutdown


if __name__ == '__main__':
    set_rabbit_credentials('daniel', 'passs')
    set_context()
    host = create_host("amqp://127.0.0.1:1679")

    e1 = host.lookup_url("amqp://127.0.0.1:1277/echo1", 'Echo', 's1_server')

    e1.echo("Hi there!")     # TELL message
    e1.echo("See ya!")

    shutdown()
```

and `pyactor\examples\Remote\s1_serverrbb.py`:

```
"""
Basic remote example sending tell messages. SERVER
@author: Daniel Barcelona Pons
"""
from pyactor.context import \
    set_context, create_host, set_rabbit_credentials, serve_forever


class Echo(object):
    _tell = {'echo'}

    def echo(self, msg):
        print(msg)


if __name__ == '__main__':
    # set_rabbit_credentials('daniel', 'passs')
    set_context()
    host = create_host("amqp://127.0.0.1:1277/")

    e1 = host.spawn('echo1', Echo)
    serve_forever()
```

You can configure your rabbit credentials with:

```
setRabbitCredentials('user', 'password')
```

If you don't, it will use the default Rabbit guest user, which only can connect locally.

2.3 Main Code

PyActor: Python Actor library

2.3.1 Context

See source

class pyactor.context.**Host**(*url*)

Host must be created using the function *create_host()*. Do not create a Host directly.

Host is a container for actors. It manages the spawn and elimination of actors and their communication through channels. Also configures the remote points where the actors will be able to receive and send queries remotely. Additionally, controls the correct management of its actors' threads and intervals.

The host is managed as an actor itself so you interact with it through its *Proxy*. This allows you to pass it to another host to spawn remotely.

Parameters **url** (*str.*) – URL that identifies the host and where to find it.

attach_interval(*interval_id*, *interval_event*)

Registers an interval event to the host.

detach_interval(*interval_id*)

Deletes an interval event from the host registry.

dumps(*param*)

Checks the parameters generating new proxy instances to avoid query concurrences from shared proxies and creating proxies for actors from another host.

has_actor(*aid*)

Checks if the given id is used in the host by some actor.

Parameters **aid** (*str.*) – identifier of the actor to check.

Returns True if the id is used within the host.

loads(*param*)

Checks the return parameters generating new proxy instances to avoid query concurrences from shared proxies and creating proxies for actors from another host.

lookup(*aid*)

Gets a new proxy that references to the actor of this host (only actors in this host) identified by the given ID.

This method can be called remotely synchronously.

Parameters **aid** (*str.*) – identifier of the actor you want.

Returns *Proxy* to the actor required.

Raises NotFoundError if the actor does not exist.

Raises HostDownError if the host is down.

lookup_url(*url*, *klass*, *module=None*)

Gets a proxy reference to the actor indicated by the URL in the parameters. It can be a local reference or a remote direction to another host.

This method can be called remotely synchronously.

Parameters

- **url** (*srt.*) – address that identifies an actor.
- **klass** (*class*) – the class of the actor.
- **module** (*srt.*) – if the actor class is not in the calling module, you need to specify the module where it is here. Also, the *klass* parameter change to be a string.

Returns *Proxy* of the actor requested.

Raises `NotFoundError`, if the URL specified do not correspond to any actor in the host.

Raises `HostDownError` if the host is down.

Raises `HostError` if there is an error looking for the actor in another server.

new_parallel(*a_function*, **params*)
Register a new thread executing a parallel method.

spawn(*aid*, *klass*, **param*, ***kparam*)
This method creates an actor attached to this host. It will be an instance of the class *klass* and it will be assigned an ID that identifies it among the host.

This method can be called remotely synchronously.

Parameters

- **aid** (*str.*) – identifier for the spawning actor. Unique within the host.
- **klass** (*class*) – class type of the spawning actor. If you are spawning remotely and the class is not in the server module, you must specify here the path to that class in the form 'module.py/Class' so the server can import the class and create the instance.
- **param** – arguments for the init function of the spawning actor class.
- **kparam** – arguments for the init function of the spawning actor class.

Returns *Proxy* to the spawned actor.

Raises `AlreadyExistsError`, if the ID specified is already in use.

Raises `HostDownError` if the host is not initiated.

stop_actor(*aid*)
This method removes one actor from the Host, stopping it and deleting all its references.

Parameters **aid** (*str.*) – identifier of the actor you want to stop.

pyactor.context.**create_host**(*url='local://local:6666/host'*)
This is the main function to create a new Host to which you can spawn actors. It will be set by default at local address if no parameter *url* is given. This function should be called once for execution or after calling `shutdown()` to the previous host.

However, it is possible to create locally more than one host and simulate a remote communication between them if they are of some remote type (*http* or *amqp*), but the first one created will be the main host, which is the one hosting the queries from the main function. Of course, every host must be initialized with a different URL(port). Although that, more than one host should not be required for any real project.

Parameters **url** (*str.*) – URL where to start and bind the host.

Returns *Proxy* to the new host created.

Raises Exception if there is a host already created with that URL.

pyactor.context.**interval**(*host*, *time*, *actor*, *method*, **args*, ***kwargs*)
Creates an Event attached to the host for management that will execute the *method* of the *actor* every *time* seconds.

See example in *Sample 11 - Intervals*

Return type

Parameters

- **host** (Proxy) – host that will manage the interval, commonly the host of the actor.
- **time** (*float*) – seconds for the intervals.
- **actor** (Proxy) – actor to which make the call every *time* seconds.
- **method** (*Str.*) – method of the *actor* to be called.
- **args** (*list*) – arguments for *method*.

Returns Event instance of the interval.

pyactor.context.**later**(*timeout*, *actor*, *method*, **args*, ***kwargs*)
Sets a timer that will call the *method* of the *actor* past *timeout* seconds.

See example in *Sample 11 - Intervals*

Parameters

- **timeout** (*int*) – seconds until the method is called.
- **actor** (Proxy) – actor to which make the call after *time* seconds.
- **method** (*Str.*) – method of the *actor* to be called.
- **args** (*list*) – arguments for *method*.

Returns manager of the later (Timer in thread, Greenlet in green_thread)

pyactor.context.**serve_forever**()
This allows the host (main host) to keep alive indefinitely so its actors can receive queries at any time. The main thread stays blocked forever. To kill the execution, press Ctrl+C.

See usage example in *Sample 6 - self.id, proxy and host*.

pyactor.context.**set_context**(*module_name='thread'*)
This function initializes the execution context deciding which type of threads are being used: classic python threads or green threads, provided by Gevent.

This should be called first of all in every execution, otherwise, the library would not work.

The default module is 'thread'.

Parameters **module_name** (*str.*) – Name of the module you want to use ('thread' or 'green_thread').

pyactor.context.**set_rabbit_credentials**(*user*, *password*)
If you use a RabbitMQ server and want to make remote queries, you might need to specify new credentials for connection.

By default, PyActor uses the guest RabbitMQ user.

Parameters

- **user** (*str.*) – Name for the RabbitMQ user.
- **password** (*str.*) – Password for the RabbitMQ user.

pyactor.context.**shutdown**(*url=None*)
Stops the Host passed by parameter or all of them if none is specified, stopping at the same time all its actors. Should be called at the end of its usage, to finish correctly all the connections and threads.

pyactor.context.**sleep**(*seconds*)
Facade for the sleep function. Do not use time.sleep if you are running green threads.

2.3.2 Proxy

See source

class pyactor.proxy.**AskRefWrapper**(*channel*, *method*, *actor_url*)
Wrapper for Ask queries that have a proxy in parameters or returns.

class pyactor.proxy.**AskWrapper**(*channel*, *method*, *actor_url*)
Wrapper for Ask type queries to the proxy. Calling it blocks the execution until the result is returned or timeout is reached. You can add the tagged parameter "timeout" to change the time limit to wait. Default timeout is set to 10s. It is also possible to specify "future=True" to get an instant response with a `Future` object with which you can manage the result.

Parameters

- **channel** (*Channel*) – communication way for the query.
- **method** (*str.*) – name of the method this query is gonna invoke.
- **actor_url** (*str.*) – URL address where the actor is set.

class pyactor.proxy.**Proxy**(*actor*)
Proxy is the class that supports to create a remote reference to an actor and invoke its methods. All the references to actors will be proxies, even the host. To get a proxy to an Actor, you should use one of the host functions that provide one, like *spawn()* or *lookup_url()*.

Parameters **actor** (Actor) – the actor the proxy will manage.

get_id()

Returns the id of the actor that this proxy holds.

Raises Exception if the proxy holds a remote actor. Use URL.

get_url()

Returns the URL of the actor that this proxy holds.

class pyactor.proxy.**TellRefWrapper**(*channel*, *method*, *actor_url*)
Wrapper for Tell queries that have a proxy in parameters.

class pyactor.proxy.**TellWrapper**(*channel*, *method*, *actor_url*)
Wrapper for Tell type queries to the proxy. Creates the request and sends it through the channel.

Parameters

- **channel** (*Channel*) – communication way for the query.
- **method** (*str.*) – name of the method this query is going to invoke.
- **actor_url** (*str.*) – URL address where the actor is set.

2.3.3 Util

Defined constants: FROM, TO, TYPE, METHOD, PARAMS, FUTURE, ASK, TELL, SRC, CHANNEL, CALLBACK, ASK_RESPONSE, FUTURE_RESPONSE, RESULT, RPC_ID

2.3.4 Remote Solutions

class pyactor.rpcserver.**RequestHandler**(*request*, *client_address*, *server*)

class pyactor.rpcserver.**Sink**(*url*)
Facade for XMLRPC proxies.

class pyactor.rpcserver.**Source**(*addr*)
Facade for simple remote communication using XMLRPCServer.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class pyactor.rabbitserver.**Sink**(*url*)
Facade for RabbitMQ concrete connexions to remote actors. This manages the publish to queues.

class pyactor.rabbitserver.**Source**(*addr*)
Facade for simple remote communication using RabbitMQ. This connection uses by default the guest RabbitMQ user. To change credentials see `setRabbitCredentials()`.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

2.3.5 Exceptions

PyActor exceptions.

exception pyactor.exceptions.**AlreadyExistsError**(*value='Not specified'*)
Actor ID repeated.

exception pyactor.exceptions.**FutureError**(*value='Not specified'*)
Some problem with the Future.

exception pyactor.exceptions.**HostDownError**
The Host is down.

exception pyactor.exceptions.**HostError**(*value='Not specified'*)
Some problem with the Host.

exception pyactor.exceptions.**IntervalError**(*value='Not specified'*)
Some problem with the interval.

exception pyactor.exceptions.**NotFoundError**(*value='Not specified'*)
Actor not found in Host.

exception pyactor.exceptions.**PyActorTimeoutError**(*method='Not specified'*)
Wait time expired.

2.4 Threading Type

2.4.1 Actor

<See source>

class pyactor.thread.actor.**Actor**(*url*, *klass*, *obj*)

Actor is the instance of an object to which is possible to access and invoke its methods remotely. Main element of the model. The host is the one to create them (spawning -> see *spawn()*).

Parameters

- **url** (*str.*) – URL where the actor is running.
- **klass** (*class*) – class type for the actor.
- **obj** (*klass*) – instance of the *klass* class to attach to the actor.

is_alive()

Returns (*bool.*) identifies the current state of the actor. **True** if it is running.

receive(*msg*)

The message received from the queue specifies a method of the class the actor represents. This invokes it. If the communication is an ASK, sends the result back to the channel included in the message as an ASK_RESPONSE.

If it is a FUTURE, generates a FUTURE_RESPONSE to send the result to the manager.

Parameters msg – The message is a dictionary using the constants defined in util.py (*pyactor.util*).

run()

Creates the actor thread which will process the channel queue while the actor *is_alive()*, making it able to receive queries.

class pyactor.thread.actor.**ActorRef**(*url*, *klass*, *channel=None*)

ActorRef contains the main components of an actor. These are the URL where it is located, the communication Channel and the class of the actor as also the synchronous and asynchronous methods the class implements. When no channel is specified a new one will be created which is also the default procedure.

Note: This is a superclass of *Actor* and has no direct functionality.

---

class pyactor.thread.rpcactor.**RPCDispatcher**(*url*, *host*, *mode*)

This is the actor that will manage remote sends and receives with other hosts. Each host has one, configured depending on the scheme specified when created.

receive(*msg*)

The message received from the queue specifies a method of the class the actor represents. This invokes it. If the communication is an ASK, sends the result back to the channel included in the message as an ASK_RESPONSE.

If it is a FUTURE, generates a FUTURE_RESPONSE to send the result to the manager.

Parameters msg – The message is a dictionary using the constants defined in util.py (*pyactor.util*).

2.4.2 Intervals

<See source>

pyactor.thread.intervals.**interval_host**(*host*, *time*, *f*, **args*, ***kwargs*)
Creates an Event attached to the *host* that will execute the *f* function every *time* seconds.

See example in *Sample 11 - Intervals*

Parameters

- **host** (Proxy) – host proxy. Can be obtained from inside a class with self.host.
- **time** (*int*) – seconds for the intervals.
- **f** (*func*) – function to be called every *time* seconds.
- **args** (*list*) – arguments for *f*.

Returns Event instance of the interval.

pyactor.thread.intervals.**later**(*timeout*, *f*, **args*, ***kwargs*)
Sets a timer that will call the *f* function past *timeout* seconds.

See example in *Sample 11 - Intervals*

Returns Timer

pyactor.thread.intervals.**sleep**(*time*)
Facade for the sleep function. Avoid using time.sleep.

Parameters **time** (*int*) – time to sleep, in seconds. (Float for second divisions)

2.4.3 Parallel

<See source>

class pyactor.thread.parallels.**ActorParallel**(*url*, *klass*, *obj*)
Actor with parallel methods. Parallel methods are invoked in new threads, so their invocation do not block the actor allowing it to process many queries at a time. To avoid concurrence problems, this actors use Locks to guarantee its correct state.

get_lock()

Returns Lock of the actor.

receive(*msg*)
Overwriting Actor.receive(). Adds the checks and features required by parallel methods.

Parameters **msg** – The message is a dictionary using the constants defined in util.py (*pyactor.util*).

class pyactor.thread.parallels.**ParallelAskWrapper**(*method*, *actor*, *lock*)
Wrapper for ask methods that have to be called in a parallel way.

class pyactor.thread.parallels.**ParallelTellWrapper**(*method*, *actor*, *lock*)
Wrapper for tell methods that have to be called in a parallel way.

2.4.4 Future

<See source>

class pyactor.thread.future.**Future**(*fid*, *future_ref*, *manager_channel*)
Container for the result of an ask query sent asynchronously which could not be resolved yet.

Parameters **fid** (*str.*) – future ID.

add_callback(*method*)
Attaches a method that will be called when the future finishes.

Parameters **method** – A callable from an actor that will be called when the future completes. The only argument for that method must be the future itself from which you can get the result though *future.:meth:'result()'*. If the future has already completed, then the callable will be called immediately.

Note: This functionality only works when called from an actor, specifying a method from the same actor.

---

done()
Return True if the future finished executing.

exception(*timeout=None*)
Return a exception raised by the call that the future represents. :param timeout: The number of seconds to wait for the exception

if the future has not been completed. None, the default, sets no limit.

Returns The exception raised by the call that the future represents or None if the call completed without raising.

Raises TimeoutError: If the timeout is reached before the future ends execution.

result(*timeout=None*)
Returns the result of the call that the future represents.

Parameters **timeout** – The number of seconds to wait for the result if the future has not been completed. None, the default, sets no limit.

Returns The result of the call that the future represents.

Raises TimeoutError: If the timeout is reached before the future ends execution.

Raises Exception: If the call raises the Exception.

running()
Return True if the future is currently executing.

send_work()
Sends the query to the actor for it to start executing the work.

It is possible to execute once again a future that has finished if necessary (overwriting the results), but only one execution at a time.

set_exception(*exception*)
Sets the result of the future as being the given exception. Only called internally.

set_result(*result*)
Sets the return value of work associated with the future. Only called internally.

class pyactor.thread.future.**FutureManager**
A manager that controls the creation and execution of the futures in a host.

class pyactor.thread.future.**FutureRef**(*fid*, *future_ref*, *manager_channel*)

result(*timeout=None*)
Returns the result of the call that the future represents.

Parameters **timeout** – The number of seconds to wait for the result if the future has not been completed. None, the default, sets no limit.

Returns The result of the call that the future represents.

Raises TimeoutError: If the timeout is reached before the future ends execution.

Raises Exception: If the call raises the Exception.

2.5 Gevent Type

2.5.1 Actor

<See source>

class pyactor.green_thread.actor.**Actor**(*url*, *klass*, *obj*)
Actor is the instance of an object to which is possible to access and invoke its methods remotely. Main element of the model. The host is the one to create them (spawning -> see *spawn()*).

Parameters

- **url** (*str.*) – URL where the actor is running.
- **klass** (*class*) – class type for the actor.
- **obj** (*klass*) – instance of the *klass* class to attach to the actor.

is_alive()

Returns (*bool.*) identifies the current state of the actor. **True** if it is running.

receive(*msg*)
The message received from the queue specifies a method of the class the actor represents. This invokes it. If the communication is an ASK, sends the result back to the channel included in the message as an ASK_RESPONSE.

If it is a FUTURE, generates a FUTURE_RESPONSE to send the result to the manager.

Parameters **msg** – The message is a dictionary using the constants defined in util.py (*pyactor.util*).

run()
Creates the actor thread which will process the channel queue while the actor *is_alive()*, making it able to receive queries.

class pyactor.green_thread.actor.**ActorRef**(*url*, *klass*, *channel=None*)
ActorRef contains the main components of an actor. These are the URL where it is located, the communication `Channel` and the class of the actor as also the synchronous and asynchronous methods the class implements. When no channel is specified a new one will be created which is also the default procedure.

Note: This is a superclass of *Actor* and has no direct functionality.

---

class pyactor.green_thread.rpcactor.**RPCDispatcher**(*url*, *host*, *mode*)
This is the actor that will manage remote sends and receives with other hosts. Each host has one, configured depending on the scheme specified when created.

receive(*msg*)
The message received from the queue specifies a method of the class the actor represents. This invokes it. If the communication is an ASK, sends the result back to the channel included in the message as an ASK_RESPONSE.

If it is a FUTURE, generates a FUTURE_RESPONSE to send the result to the manager.

Parameters **msg** – The message is a dictionary using the constants defined in util.py (*pyactor.util*).

2.5.2 Intervals

<See source>

pyactor.green_thread.intervals.**interval_host**(*host*, *time*, *f*, **args*, ***kwargs*)
Creates an Event attached to the *host* that will execute the *f* function every *time* seconds.

See example in *Sample 11 - Intervals*

Parameters

- **host** (Proxy) – host proxy. Can be obtained from inside a class with `self.host`.
- **time** (*int*) – seconds for the intervals.
- **f** (*func*) – function to be called every *time* seconds.
- **args** (*list*) – arguments for *f*.

Returns Event instance of the interval.

pyactor.green_thread.intervals.**later**(*timeout*, *f*, **args*, ***kwargs*)
Sets a timer that will call the *f* function past *timeout* seconds.

See example in *Sample 11 - Intervals*

Returns Greenlet new 'thread' which will perform the call when specified.

pyactor.green_thread.intervals.**sleep**(*seconds*)
Facade for the sleep function. Do not use time.sleep if you are running green threads.

Parameters **time** (*int*) – time to sleep, in seconds. (Float for second divisions)

2.5.3 Parallel

<See source>

class pyactor.green_thread.parallels.**ActorParallel**(*url*, *klass*, *obj*)
Actor with parallel methods. Parallel methods are invoked in new threads, so their invocation do not block the actor allowing it to process many queries at a time. Green threads do not have concurrence problems so no need to use Locks in this implementation.

receive(*msg*)
Overwriting Actor.receive(). Adds the checks and features required by parallel methods.

Parameters **msg** – The message is a dictionary using the constants defined in util.py (*pyactor.util*).

class pyactor.green_thread.parallels.**ParallelAskWrapper**(*method*, *actor*)
Wrapper for ask methods that have to be called in a parallel way.

class pyactor.green_thread.parallels.**ParallelTellWrapper**(*method*, *actor*)
Wrapper for tell methods that have to be called in a parallel way.

2.5.4 Future

<See source>

class pyactor.green_thread.future.**Future**(*fid*, *future_ref*, *manager_channel*)
Container for the result of an ask query sent asynchronously which could not be resolved yet.

Parameters **fid** (*str.*) – future ID.

add_callback(*method*)
Attaches a method that will be called when the future finishes.

Parameters **method** – A callable from an actor that will be called when the future completes. The only argument for that method must be the future itself from which you can get the result though *future.:meth:'result()'*. If the future has already completed, then the callable will be called immediately.

Note: This functionality only works when called from an actor, specifying a method from the same actor.

---

done()
Return True if the future finished executing.

exception(*timeout=None*)
Return a exception raised by the call that the future represents. :param timeout: The number of seconds to wait for the exception

if the future has not been completed. None, the default, sets no limit.

Returns The exception raised by the call that the future represents or None if the call completed without raising.

Raises TimeoutError: If the timeout is reached before the future ends execution.

result(*timeout=None*)
Returns the result of the call that the future represents.

Parameters **timeout** – The number of seconds to wait for the result if the future has not been completed. None, the default, sets no limit.

Returns The result of the call that the future represents.

Raises TimeoutError: If the timeout is reached before the future ends execution.

Raises Exception: If the call raises the Exception.

running()
Return True if the future is currently executing.

send_work()
Sends the query to the actor for it to start executing the work.

It is possible to execute once again a future that has finished if necessary (overwriting the results), but only one execution at a time.

set_exception(*exception*)
Sets the result of the future as being the given exception. Only called internally.

set_result(*result*)
Sets the return value of work associated with the future. Only called internally.

class pyactor.green_thread.future.**FutureManager**
A manager that controls the creation and execution of the futures in a host.

class pyactor.green_thread.future.**FutureRef**(*fid*, *future_ref*, *manager_channel*)

result(*timeout=None*)
Returns the result of the call that the future represents.

Parameters **timeout** – The number of seconds to wait for the result if the future has not been completed. None, the default, sets no limit.

Returns The result of the call that the future represents.

Raises TimeoutError: If the timeout is reached before the future ends execution.

Raises Exception: If the call raises the Exception.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

P

R

S

T