
uactor

Release 0.1.1

Felipe A Hernandez

Jul 23, 2020

CONTENTS:

1	uActor: Process Actor Model	3
1.1	uActor: Process Actor Model	3
1.2	uactor	9
1.3	License	12
1.4	Actor inheritance	13
1.5	Actor lifetime	14
1.6	Result proxies	15
1.7	Method callbacks	17
1.8	Sticky processes	18
1.9	Actor pool example	18
1.10	Networking	20
2	Indices and tables	25
	Python Module Index	27
	Index	29

Welcome to uActor documentation.

UACTOR: PROCESS ACTOR MODEL

uActor is a process actor library for Python with a simple yet powerful API, implementing the [actor model](#) atop [multiprocessing](#), with no dependencies other than the [Python Standard Library](#).



1.1 uActor: Process Actor Model

uActor is a process actor library for Python with a simple yet powerful API, implementing the [actor model](#) atop [multiprocessing](#), with no dependencies other than the [Python Standard Library](#).

- **Simple:** Minimalistic API, no boilerplate required.
- **Flexible:** Trivial to integrate, meant to be extended.
- **Concurrent:** Share workload over CPU cores, and across the network.

Documentation: uactor.readthedocs.io

Usage:

```
import os
import uactor

class Actor(uactor.Actor):
    def hello(self):
        return f'Hello from subprocess {os.getpid()}!'

print(f'Hello from process {os.getpid()}!')
# Hello from process 22682!

print(Actor().hello())
# Hello from subprocess 22683!
```

1.1.1 Quickstart

Installation

You can install it using `pip`.

```
pip install uactor
```

Or alternatively by including our single `uactor.py` file into your project.

Your first actor

With `uActor`, actors are defined as classes inheriting from `uactor.Actor`, with some special attributes we'll cover later.

```
import uactor

class MyActor(uactor.Actor):
    def my_method(self):
        return True
```

During instantiation, every actor is initialized on its own dedicated process, returning a proxy.

```
my_actor_proxy = MyActor()
my_actor_proxy.my_method()
```

Once you're done with your actor, it is always a good idea to finalize its process with `uactor.Actor.shutdown` method.

```
my_actor_proxy.shutdown()
```

Alternatively, `uactor.Actor` instances can be used as context managers, so the actor process will be finalized once we're done with it.

```
with MyActor() as my_actor_proxy:
    my_actor_proxy.my_method()
```

Actor processes will be also finished when every proxy gets garbage-collected on its parent process.

Returning result proxies

Actor methods can return proxies instead of actual objects, keeping them sound and safe on our actor process.

To specify which proxy will be returned from an specific method, we can add both method name and proxy typeid to `uactor.Actor._method_to_typeid_` special class attribute.

```
import uactor

class MyActor(uactor.Actor):

    _method_to_typeid_ = {'my_method': 'dict'}

    def __init__(self):
        self.my_data = {}
```

(continues on next page)

(continued from previous page)

```
def my_method(self):
    return self.my_data
```

Or, alternatively, we can explicitly create a proxy for our object, using `uactor.proxy` utility function.

```
import uactor

class MyActor(uactor.Actor):
    def __init__(self):
        self.my_data = {}

    def my_method(self):
        return uactor.proxy(self.my_data, 'dict')
```

There is a limitation with proxies, applying two different proxies to the same object will raise an exception, this is likely to change in the future.

Becoming asynchronous (and concurrent)

Actor methods are fully synchronous by default, which is usually not very useful on distributed software, the following example will show you how to return asynchronous results from the actor.

```
import time
import multiprocessing.pool
import uactor

class MyActor(uactor.Actor):
    _method_to_typeid_ = {'my_method': 'AsyncResult'}

    def __init__(self):
        self.threadpool = multiprocessing.pool.ThreadPool()

    def my_method(self):
        return self.threadpool.apply_async(time.sleep, [10]) # wait 10s

with MyActor() as my_actor:

    # will return immediately
    result = my_actor.my_method()

    # will take 10 seconds
    result.wait()
```

Based on this, we can now run code concurrently running on the same actor.

```
with MyActor() as my_actor:

    # these will return immediately
    result_a = my_actor.my_method()
    result_b = my_actor.my_method()

    # these all will take 10 seconds in total
    result_a.wait()
    result_b.wait()
```

And now we can to parallelize workloads across different actor processes.

```
actor_a = MyActor()
actor_b = MyActor()
with actor_a, actor_b:

    # these both will return immediately
    result_a = actor_a.my_method()
    result_b = actor_b.my_method()

result_a.wait() # this will take ~10s to complete
result_b.wait() # this will be immediate (we already waited 10s)
```

Next steps

You can dive into our [documentation](#) or take a look at our code examples.

- The basics:
 - *Actor inheritance.*
 - *Actor lifetime.*
 - *Result proxies.*
 - *Method callbacks.*
- Advanced patterns:
 - *Sticky processes.*
 - *Actor pool.*
 - *Networking.*

1.1.2 uActor design

With the constant rise in CPU core count, highly threaded python applications are still pretty rare (except for distributed processing frameworks like [celery](#)), this is due a few reasons:

- [threading](#) cannot use multiple cores because [Python Global Interpreter Lock](#) forces the interpreter to run on a single core.
- [multiprocessing](#), meant to overcome threading limitations by using processes, exposes a pretty convoluted API as processes are way more complex, exposing many quirks and limitations.

uActor allows implementing distributed software as easy as just declaring and instancing classes, following the [actor model](#), by thinly wrapping the standard [SyncManager](#) to circumvent most o [multiprocessing](#) complexity and some of its flaws.

uActor API is designed to be both minimalistic and intuitive, but still few compromises had to be taken to leverage on [SyncManager](#) as much as possible, as it is both somewhat actively maintained and already available as part of the [Python Standard Library](#).

Actors

Just like the actor programming model revolves around the actor entity, uActor features the `uactor.Actor` base class.

When an actor class is declared, by inheriting from `uactor.Actor`, its `Actor.proxy_class` gets also inherited and updated to mirror the actor interface, either following the explicit list of properties defined at `Actor._exposed_` or implicitly by actor public methods.

`Actor.manager_class` is also inherited registering actor specific proxies defined in `Actor._proxies_` mapping (key used as a typeid) along with 'actor' and 'auto' special proxies.

Keep in mind the default `Actor.manager_class`, `uactor.ActorManager`, already includes every proxy from [SyncManager](#) (including the internal `AsyncResult` and `Iterator`) which are all available to the actor and ready use (you can call `Actor.manager_class.typeids()` to list them all).

As a reference, these are all the available `uactor.Actor` configuration class attributes:

- `manager_class`: manager base class (defaults to parent's one, up to `uactor.ActorManager`).
- `proxy_class`: actor proxy class (defaults to parent's one, up to `uactor.ActorProxy`).
- `_options_`: option mapping will be passed to `manager_class`.
- `_exposed_`: list of explicitly exposed methods will be made available by `proxy_class`, if `None` or undefined then all public methods will be exposed.
- `_proxies_`: mapping (typeid, proxy class) of additional proxies will be registered in the `manager_class` and, thus, will be available to be returned by the actor.
- `_method_to_typeid_`: mapping (method name, typeid) defining which method return values will be wrapped into proxies when invoked from `proxy_class`.

When an `uactor.Actor` class is instantiated, a new process is spawned and a `uactor.Actor.proxy_class` instance is returned (as the real actor will be kept safe in said process), transparently exposing a message-based interface.

```
import os
import uactor

class Actor(uactor.Actor):
    def getpid(self):
        return os.getpid()

actor = Actor()
print('My process id is', os.getpid())
# My process id is 153333
print('Actor process id is ', actor.getpid())
# Actor process id is 153344
```

Proxies

Proxies are objects communicating with the actor process, exposing a similar interface, in the most transparent way possible.

It is implied most calls made to a proxy will result on inter-process communication and serialization overhead.

To alleviate the serialization cost, actor methods can also return proxies, so the real data is kept well inside the actor process boundaries, which can be efficiently shared between processes with very little serialization cost.

Actors can define which proxy will be used to expose the result of certain methods by defining that in their `Actor._method_to_typeid_` property.

```
import uactor

class Actor(uactor.Actor):
    _method_to_typeid_ = {'get_mapping': 'dict'}
    ...
    def get_data(self):
        return self.my_data_dict
```

Or, alternatively, using the `uactor.proxy` function, receiving both value and a proxy `typeid` (as in `SyncManager` semantics).

```
import uactor

class Actor(uactor.Actor):
    ...
    def get_data(self):
        return uactor.proxy(self.my_data_dict, 'dict')
```

Keep in mind `uactor.proxy` can only be called from inside the actor process (it will raise `uactor.ProxyError` otherwise), as proxies can only be created from there.

You can define your own proxy classes (following `BaseProxy` semantics), and they will be made available in an actor by including it on the `Actor._proxies_` mapping (along its `typeid`).

```
import uactor

class MyDataProxy(uactor.BaseProxy):
    def my_method(self):
        return self._callmethod('my_method')

    my_other_method = uactor.ProxyMethod('my_other_method')

class Actor(uactor.Actor):
    _proxies_ = {'MyDataProxy': MyDataProxy}
    _method_to_typeid_ = {'get_data': 'MyDataProxy'}
    ...
```

In addition to all proxies imported from both `SyncManager` (including internal ones as `Iterator` and `AsyncResult`) and `Actor._proxies_`, we always register these ones:

- `actor`: proxy to the current process actor.
- `auto`: dynamic proxy based based on the wrapped object.

You can list all available proxies (which can vary between python versions) by calling `ActorManager.typeids()`:

```
import uactor

print(uactor.Actor.manager_class.typeids())
# ('Queue', 'JoinableQueue', 'Event', ..., 'auto', 'actor')

print(uactor.ActorManager.typeids())
# ('Queue', 'JoinableQueue', 'Event', ..., 'auto')
```

1.1.3 Contributing

uActor is deliberately very small in scope, while still aiming to be easily extended, so extra functionality might be implemented via external means.

If you find any bug or a possible improvement to existing functionality it will likely be accepted so feel free to contribute.

If, in the other hand, you feel a feature is missing, you can either create another library using uActor as dependency or fork this project.

1.1.4 License

Copyright (c) 2020, Felipe A Hernandez.

MIT License (see [LICENSE](#)).

1.2 uactor

uActor module.

uactor.DEFAULT_SERIALIZER = 'pickle'
Default `multiprocessing.managers` serializer name for *uactor*.

New in version 0.1.1.

exception uactor.UActorException
Bases: `Exception`

Base exception for uactor module.

New in version 0.1.0.

exception uactor.ProxyError
Bases: `uactor.UActorException`

Exception for errors on proxy logic.

New in version 0.1.0.

exception uactor.AuthkeyError
Bases: `uactor.ProxyError`

Exception raised when connecting to proxy with invalid authkey.

New in version 0.1.1.

```
class uactor.BaseProxy(token, serializer, manager=None, authkey=None, exposed=None, incref=True, manager_owned=False)
    Bases: multiprocessing.managers.BaseProxy
```

Base Proxy class.

This class implements a few workarounds around bugs found in `multiprocessing.managers.BaseProxy` by preventing `BaseProxy._manager` from getting lost on both unserialization and process forking by recreating it on demand.

New in version 0.1.0.

```
class uactor.ActorManager(address: Optional[AddressType] = None, authkey: Optional[bytes] = None, serializer: str = 'pickle', *args, parent: Optional[ActorManager] = None, **kwargs)
    Bases: multiprocessing.managers.BaseManager
```

Multiprocessing manager for uactor.

New in version 0.1.0.

```
classmethod typeids() → Tuple[str, ...]
    Get tuple of typeid of all registered proxies.
```

```
property process
    Get remote actor process if owned by this manager.
```

```
start (*args, **kwargs)
    Start manager process.
```

```
connect ()
    Connect to manager process.
```

Raises **AuthkeyError** – on actor process authkey rejection.

```
class uactor.ActorProxy(token, serializer, manager=None, authkey=None, exposed=None, incref=True, manager_owned=False)
    Bases: uactor.BaseProxy
```

Actor proxy base class.

Actors will inherit from this class to create custom implementations based on their declared configuration and interface.

New in version 0.1.0.

```
property connection_address
    Get connection address to Actor process.
```

New in version 0.1.1.

```
__enter__ (*args, **kwargs)
    Call Actor.__enter__() method.
```

```
__exit__ (*args, **kwargs)
    Call Actor.__exit__() method.
```

When this proxy is the direct result from instantiating the *Actor* class, calling this method will also result on *Actor.shutdown()* being called, finishing the actor process (like when calling *ActorProxy.shutdown()*).

```
shutdown ()
    Call Actor.shutdown() method.
```

When the current process is responsible of initializing the actor, calling this method will also finish the actor process.

```
class uactor.Actor(*args, **kwargs)
```

Bases: `object`

Actor base class for actor implementations to inherit from.

An actor represents a processing unit. During instantiation, a new actor process is be started, and the corresponding proxy is returned.

Actors also implement the context manager interface, and you can override both `Actor.__enter__()` and `Actor.__exit__()` to implement your own logic, but keep in mind they're both specially handled and calling `ActorProxy.__exit__()` will also terminate the process (just like calling `ActorProxy.shutdown()`).

New in version 0.1.0.

manager_class

`ActorManager` subclass used to initialize the actor process.

Whatever is defined here, will be subclassed during actor class initialization to apply the declared actor configuration.

alias of `ActorManager`

proxy_class

`ActorProxy` subclass used to initialize the actor proxy.

Whatever is defined here, will be subclassed during actor class initialization to apply the declared actor configuration.

alias of `ActorProxy`

```
_options_: Mapping[str, Any] = {}
```

Option `dict` will be passed to `Actor.manager_class`.

This options mapping is passed to `Actor.manager_class` during `Actor` instantiation.

```
_exposed_: Optional[Tuple[str]] = ('__enter__', '__exit__', 'shutdown')
```

`tuple` containing then list of method/properties will be exposed.

Class inheritance will be honored when using this attribute.

```
_proxies_: Mapping[str, Type[uactor.BaseProxy]] = {'Array': <class 'uactor.Proxy[Arr
```

Proxy `dict` of typeid keys and `BaseProxy` values.

Proxies defined here will be registered at `Actor.manager_class` and will be made available from within the actor process.

```
_method_to_typeid_: Mapping[str, str] = {'__enter__': 'actor'}
```

Configuration `dict` of method name keys and typeid values.

Including a method with an typeid here will result on the corresponding proxy to be returned when called from an `ActorProxy` instance.

```
__enter__() → TActor
```

Enter context, return actor proxy.

```
__exit__(exc_type: Type[Exception], exc_val: Exception, exc_tb: traceback) → Optional[bool]
```

Leave context.

Method `uactor.Actor.shutdown()` will be called after this one when the context manager interface is used on the owner process.

shutdown()

Perform shutdown work before the process dies (stub).

This method will be called by explicit `ActorProxy.shutdown()` calls, even if no real process shutdown is involved (ie. with remote proxy connections), enabling implementing remote shutdown logic here (ie. breaking a mainloop).

This method will be also called after `Actor.__exit__()` when the owner process uses the `ActorProxy` context manager interface.

classmethod connect (*address: Union[Tuple[str, int], str, bytes, int], authkey: bytes, serializer: str = 'pickle', capture: Sequence[Union[Tuple[str, int], str, bytes, int]] = ()*)
→ TActorProxy

Get actor proxy instance connected to address.

Parameters

- **address** – actor process connection address
- **authkey** – authentication secret key
- **serializer** – serializer name
- **capture** – iterable of additional addresses will be handled with this connection.

New in version 0.1.1.

`uactor.proxy` (*value: Any, typeid: str = 'auto', serializer: str = 'pickle'*) → `uactor.BaseProxy`
Create serialized proxy from given value and typeid (defaults to `auto`).

This function can be only used from inside the actor process.

New in version 0.1.0.

`uactor.typeid` (*proxy: uactor.BaseProxy*) → `str`
Get typeid from given proxy object.

New in version 0.1.0.

1.3 License

MIT License

Copyright (c) 2020 Felipe A Hernandez <ergoithz@gmail.com>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.4 Actor inheritance

Actor inheritance works just as regular python inheritance (just a few caveats on special attributes, see below).

```
import os
import uactor

class Feline(uactor.Actor):

    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"[{os.getpid()}] Hi, it's {self.name}."

class Cat(Feline):

    def greet(self):
        return f'{super().greet()} Meow.'

class Tiger(Feline):

    def greet(self):
        return f'{super().greet()} Roar.'

cat = Cat('Mr. Whiskers')
tiger = Tiger('Mr. Fangs')

print(f"[{os.getpid()}] Hello everyone.")
# [297381] Hello everyone.

print(cat.greet())
# [299145] Hi, it's Mr. Whiskers. Meow.

print(tiger.greet())
# [299165] Hi, it's Mr. Fangs. Roar.
```

1.4.1 Configuration inheritance

Actor configuration attributes `_exposed_`, `_proxies_` and `_method_to_typeid_` are inheritance-aware (that is, all parent values are honored), so you don't need to carry parent values manually when updating them.

```
import uactor

class Parent(uactor.Actor):

    _exposed_ = ('greet',)

    def greet(self):
        return f"It's {type(self).__name__}."

    def private(self):
        return "This method won't be available in the proxy"

class Child(Parent):
```

(continues on next page)

(continued from previous page)

```

    _exposed_ = ('hello',)

    def hello(self):
        return f'{super().greet()} Hello.'

print(Parent().greet())
# It's Parent.

print(Child().greet())
# It's Child.

print(Child().hello())
# It's Child. Hello.

try:
    print(Parent().private())
except AttributeError as e:
    print(e)
    # 'Parent.proxy_class' object has no attribute 'private'

try:
    print(Child().private())
except AttributeError as e:
    print(e)
    # 'Child.proxy_class' object has no attribute 'private'

```

1.5 Actor lifetime

It is always advised to hold external resources only as long as they're needed, freeing them after that, and actors are not an exception to this.

Actors expose both [context manager protocol](#) and shutdown methods to enable finalizing the actor process once is no longer required.

```

import uactor

class Actor(uactor.Actor):

    def __init__(self):
        print('Initialized')

    def __enter__(self):
        print('Context enter')
        return super().__enter__() # return actor proxy

    def __exit__(self, exc_type, exc_value, traceback):
        print('Context exit')
        return super().__exit__(exc_type, exc_value, traceback) # shutdown

    def shutdown(self):
        print('Shutdown')

with Actor() as actor:

```

(continues on next page)

(continued from previous page)

```

    # Initialized
    # Context enter
    pass
# Context exit
# Shutdown

actor = Actor()
# Initialized
actor.shutdown()
# Shutdown

```

If you forget to manually finish the actor, don't worry, actor processes will be also finished when all their proxies get garbage-collected on its parent process.

1.6 Result proxies

UActor provide two different ways to return proxies to objects living inside the actor process: declarative and wrapping, supporting different use-cases:

- Defining the method proxy via `uactor.Actor._method_to_typeid_` results in the specified proxy to be returned only when called from actor proxy, so calls from within the actor itself will still receive the actual result.
- Using `uactor.proxy` helper explicitly specifies a proxy from the method, so you can to dynamically choose between different proxies and return values. These proxies will only be functional when received by the main process or other actors.

```

import uactor

class Actor(uactor.Actor):
    _method_to_typeid_ = {'get_declarative_proxy_to_data': 'list'}

    def __init__(self):
        self.data = [1, 2, 3]

    def get_declarative_proxy_to_data(self):
        return self.data

    def get_serialized_proxy_to_data(self):
        return uactor.proxy(self.data, 'list')

with Actor() as actor:

    proxy = actor.get_declarative_proxy_to_data()
    print(type(proxy), uactor.typeid(proxy), list(proxy))
    # <class 'multiprocessing.managers.ListProxy'> list [1, 2, 3]

    proxy = actor.get_serialized_proxy_to_data()
    print(type(proxy), uactor.typeid(proxy), list(proxy))
    # <class 'multiprocessing.managers.ListProxy'> list [1, 2, 3]

```

1.6.1 Serialized proxies

The serialized proxy pattern is useful when you need to conditionally return different proxies or values.

When `uactor.proxy` is called, a new proxy is created for the given value and typeid, which can be transferred safely to other processes.

```
import uactor

class Actor(uactor.Actor):

    def __init__(self):
        self.data = [1, 2, 3]

    def get_data(self, as_proxy=False):
        return uactor.proxy(self.data, 'list') if as_proxy else self.data

with Actor() as actor:

    value = actor.get_data()
    print(type(value), value)
    # <class 'list'> [1, 2, 3]

    proxy = actor.get_data(as_proxy=True)
    print(type(proxy), list(proxy))
    # <class 'multiprocessing.managers.ListProxy'> [1, 2, 3]
```

1.6.2 Synchronization proxies

uActor makes quite easy to share synchronization primitives between processes, by including specific proxies for this such as Event, Lock, RLock, Semaphore, BoundedSemaphore, Condition and Barrier, which can be used with primitives from threading, or even multiprocessing (albeit using proxies to multiprocessing should be avoided):

```
import threading
import uactor

class Actor(uactor.Actor):
    _exposed_ = ('event',)

    @property
    def event(self):
        return uactor.proxy(self._event, 'Event')

    def __init__(self):
        self._event = threading.Event()

with Actor() as actor:
    print('Ready' if actor.event.wait(1) else 'Not ready')
    # Not ready

    actor.event.set()

    print('Ready' if actor.event.wait(1) else 'Not ready')
    # Ready
```

1.6.3 Asynchronous proxies

uActor includes those extremely useful `Pool` and `AsyncResult` (for `(for multiprocessing.pool.Pool)` and `Queue` (for `queue.Queue`) proxies.

This allow to parallelize work across multiple actors way easier than using raw primitives, just by sharing asynchronous result objects or queues.

```
import time
import multiprocessing.pool
import uactor

class Actor(uactor.Actor):
    _exposed_ = ('pool',)

    @property
    def pool(self):
        return uactor.proxy(self._pool, 'Pool')

    def __init__(self):
        self._pool = multiprocessing.pool.ThreadPool()

with Actor() as actor:
    start = time.time()
    async_result = actor.pool.apply_async(time.sleep, (2,))
    print(f'round(time.time() - start, 4) }s')
    # 0.0014s

    async_result.get()
    print(f'round(time.time() - start, 4) }s')
    # 2.0032s
```

1.7 Method callbacks

One common pattern in the actor programming model is to carry the result of a method call as parameter of another one. This is called callback, and can be used in many contexts to avoid blocking the main application process while waiting for results.

This can be very useful when used along with asynchronous [result proxies](#).

```
import uactor

class ActorA(uactor.Actor):

    def send(self, target):
        return target('ping')

class ActorB(uactor.Actor):

    def receive(self, value):
        return 'pong' if value == 'ping' else 'error'

actor_a = ActorA()
actor_b = ActorB()
```

(continues on next page)

(continued from previous page)

```
print(actor_a.send(actor_b.receive))
# pong
```

1.8 Sticky processes

When handling multi-process concurrency, your operative system (or more specifically, its process scheduler) will effectively distribute the workload between processes at its best.

But, when looking for maximum performance, we may want to prevent two actors to run in the same CPU core, otherwise they have to share processing time.

Thanks to the awesome `psutil` library we can do this simply by selecting an specific CPU core per process.

```
import psutil
import uactor

class StickyActor(uactor.Actor):
    def __init__(self, core):
        # Stick our current actor process to a core
        psutil.Process().cpu_affinity([core])

# Initialize one actor per CPU core
actors = [
    StickyActor(core)
    for core in range(psutil.cpu_count())
]
```

This pattern fits very well into `actor pools` for better distributing workloads.

1.9 Actor pool example

As with every multiprocessing framework, the necessity of keeping track of many execution units (in our case, actors) is quite common with `uActor`.

Here's where object pools come to hand, allowing to keep track of many objects at the same time.

Here we will explain some approaches on implementing actor pools, taking concurrency into consideration.

1.9.1 Client parallelization

Actor pool example where parallelization is achieved at the client side using threads, calling to a synchronous actor.

```
import os
import itertools
import multiprocessing.pool
import uactor

class SyncActor(uactor.Actor):
    def getpid(self):
        return os.getpid()

class AsyncActorPool:
```

(continues on next page)

(continued from previous page)

```

def __init__(self, size, cls, *args, **kwargs):
    self.threadpool = multiprocessing.pool.ThreadPool(size)
    self.pool = [cls(*args, **kwargs) for _ in range(size)]
    self.actors = itertools.cycle(self.pool)

def call(self, method, *args, **kwargs):
    func = getattr(next(self.actors), method)
    return self.threadpool.apply_async(func, args, kwargs)

def broadcast(self, method, *args, **kwargs):
    return self.threadpool.map_async(
        lambda actor: getattr(actor, method)(*args, **kwargs),
        self.pool,
    )

def __enter__(self):
    self.threadpool.__enter__()
    self.broadcast('__enter__').wait()
    return self

def __exit__(self, exc_type, exc_val, exc_tb):
    return any([
        *self.broadcast('__exit__', exc_type, exc_val, exc_tb).get(),
        self.threadpool.__exit__(exc_type, exc_val, exc_tb),
    ])

with AsyncActorPool(4, SyncActor) as pool:
    results = [pool.call('getpid') for _ in range(5)]
    print([result.get() for result in results])

```

1.9.2 Actor asynchronous results

Actor pool example where parallelization is performed on the actor side, returning `AsyncResult` proxies (see [result proxies](#)).

```

import os
import itertools
import multiprocessing.pool
import uactor

class AsyncActor(uactor.Actor):
    _method_to_typeid_ = {'getpid': 'AsyncResult'}

    def __init__(self):
        self.threadpool = multiprocessing.pool.ThreadPool(4)

    def getpid(self):
        return self.threadpool.apply_async(os.getpid)

class SyncActorPool:
    def __init__(self, size, cls, *args, **kwargs):
        self.pool = [cls(*args, **kwargs) for _ in range(size)]
        self.actors = itertools.cycle(self.pool)

    def call(self, method, *args, **kwargs):

```

(continues on next page)

(continued from previous page)

```

        return getattr(next(self.actors), method)(*args, **kwargs)

    def __enter__(self):
        for actor in self.pool:
            actor.__enter__()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        return any([
            actor.__exit__(exc_type, exc_val, exc_tb)
            for actor in self.pool
        ])

with SyncActorPool(4, AsyncActor) as pool:
    results = [pool.call('getpid') for _ in range(5)]
    print([result.get() for result in results])

```

1.10 Networking

Actors will default to the most efficient method of inter-process communication available.

But in some cases, you may want to distribute workloads between different machines on a same network over TCP/IP. This can be done by defining the appropriate addresses on your actors.

Keep in mind that the same actors classes must be available, at the same modules, in all the involved parties.

1.10.1 Connection

We can declare our network actor as usual, but customizing the options forwarded to `ActorManager` with a TCP address (in the example below, `0.0.0.0` means listening to all addresses, while `0` means choosing a random port), we also need to specify an `authtoken` (the authentication secret), initialize the actor (listening to incoming connections), and print at which port is the actor listening at.

We'll name this module `network_actor.py` to be used later.

```

import os
import time
import uactor

class NetworkActor(uactor.Actor):

    # Actor manager options to listen over TCP on a random port
    _options_ = {'address': ('0.0.0.0', 0), 'authkey': b'SECRET'}

    def getpid(self):
        return os.getpid()

if __name__ == '__main__':
    with NetworkActor() as actor:
        host, port = actor.connection_address
        print(f'Actor process {actor.getpid()} at {host}:{port}')
        # Actor process 140262 at 0.0.0.0:37255

```

(continues on next page)

(continued from previous page)

```
while True: # keep the owner proxy alive
    time.sleep(10)
```

We can now connect, remotely, to the same actor process with the `uactor.Actor.connect` method with the correct authkey, keep in mind both proxy address hostname and port to reach its actor process can vary at different network locations.

This is a remote connection example (importing the actor class from above):

```
from network_actor import NetworkActor

address = 'localhost', 37255
with NetworkActor.connect(address, b'SECRET') as actor:
    host, port = actor.connection_address
    print(f'Actor process {actor.getpid()} at {host}:{port}')
    # Actor process 140262 at localhost:37255
```

1.10.2 Forwarded proxies

One neat feature of uActor is proxy forwarding, that is, being able to pass proxies as arguments or return them, to and from actors.

But when forwarding proxies from actors with different secrets, complexity adds up pretty quickly.

If a proxy returns a foreign proxy from an actor we aren't connected to, an `AuthkeyError` will be raised because our process does not know its authkey.

```
import uactor

class MyActor(uactor.Actor):
    _exposed_ = ('my_other_actor',)

    def __init__(self):
        self.my_other_actor = MyOtherActor()

class MyOtherActor(uactor.Actor):
    _options_ = {'address': ('0.0.0.0', 7000), 'authkey': b'OtherSecret'}

with MyActor() as actor:
    my_other_actor = actor.my_other_actor
    # AuthkeyError
```

We need to connect to actors before being able to take proxies pointing to them, while at the same time we probably need to translate those proxies addresses to be reachable from our location.

```
with MyActor() as actor:
    address = 'localhost', 7000
    capture = [('0.0.0.0', 7000)]
    with MyOtherActor.connect(address, b'OtherSecret', capture=capture):
        my_other_actor = actor.my_other_actor
```

Or, alternatively, we can wait until we get an exception to perform the connection.

```
with MyActor() as actor:
    try:
```

(continues on next page)

(continued from previous page)

```

    my_other_actor = actor.my_other_actor
except uactor.AuthKeyError as e:
    address = 'localhost', 7000
    capture = [('0.0.0.0', 7000)]
    with MyOtherActor.connect(address, b'OtherSecret', capture=capture):
        my_other_actor = actor.my_other_actor

```

Both approaches have their pros and cons, is up to the developer to choose wisely between them, based on the side-effects on his implementation.

1.10.3 Server mainloop with remote shutdown

Since the actor process have to be kept alive by its parent process, we can implement some simple logic to keep it around until needed, while allowing remote shutdowns.

We'll name this module `network_actor.py` to be used later.

```

import threading
import uactor

class NetworkActor(uactor.Actor):

    # Actor manager options to listen over TCP on a random port
    _options_ = {'address': ('0.0.0.0', 6000), 'authkey': b'SECRET'}

    def __init__(self):
        self.finished = threading.Event()

    def shutdown(self):
        self.finished.set()

    def wait(self, timeout=-1):
        return self.finished.wait(timeout)

if __name__ == '__main__':
    with NetworkActor() as actor:
        while not actor.wait(timeout=10): # avoid socket timeouts
            pass

```

This way, a remote proxy will be able to end the mainloop by calling shutdown and end the owner process mainloop, effectively finishing the process. We'll import the actor class from the previous example.

```

from network_actor import NetworkActor

address = 'localhost', 6000
external = NetworkActor.connect(address, b'SECRET')
external.shutdown()

```

1.10.4 Autodiscovery

To enable dynamic actor discovery, we might keep an central actor listening to an specific port, acting as an central registry for other actors.

This way, we can start as many actors as we like, at any time.

We'll name this module `network_actor_registry.py` to be used later.

```
import itertools
import os
import time
import uactor

class Registry(uactor.Actor):

    _options_ = {'address': ('0.0.0.0', 5000), 'authkey': b'SECRET'}

    def __init__(self):
        self.addresses = frozenset()
        self.iterator = iter(())

    def register(self, *addresses):
        addresses = self.addresses.union(addresses)
        self.iterator, self.addresses = itertools.cycle(addresses), addresses

    def pick(self):
        return next(self.iterator, None)

class NetworkActor(uactor.Actor):

    # Actor manager options to listen over TCP on a random port
    _options_ = {'address': ('0.0.0.0', 0), 'authkey': b'SECRET'}

    def getpid(self):
        return os.getpid()

if __name__ == '__main__':
    with Registry() as registry:
        actors = [NetworkActor() for actor in range(10)]
        addresses = [actor.connection_address for actor in actors]
        registry.register(*addresses)

        print(f'Registry listening at port {registry.connection_address[1]}')
        # Registry serving at port 5000

        print(f'Actors listening at ports {[port for _, port in addresses]}')
        # Actors listening at ports [36061, 35245, ..., 33701, 41653]

        while True: # keep registry and actors alive
            time.sleep(10)
```

Using registry also allow us to register new actors dynamically.

```
import time

from network_actor_registry import Registry, NetworkActor
```

(continues on next page)

(continued from previous page)

```
address = 'localhost', 5000
with Registry.connect(address) as registry:
    actors = [NetworkActor() for actor in range(10)]
    addresses = [actor.connection_address for actor in actors]
    registry.register(*addresses)

    print(f'Actors listening at ports {[port for _, port in addresses]}')
    # Actors listening at ports [36061, 35245, ..., 33701, 41653]

    while True: # keep actors alive
        time.sleep(10)
```

And we can access those actors by retrieving their addresses from the registry (taking care of handling local addresses, see *forwarded proxies*).

```
from network_actor_registry import Registry, NetworkActor

address = 'localhost', 5000
with Registry.connect(address, b'SECRET') as registry:
    for i in range(10):
        _, port = registry.pick()
        address = 'localhost', port
        with NetworkActor.connect(address, b'SECRET') as actor:
            print(f'Actor at port {port} has pid {actor.getpid()}')
```

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

U

uactor, [9](#)

Symbols

[__enter__\(\) \(uactor.Actor method\), 11](#)
[__enter__\(\) \(uactor.ActorProxy method\), 10](#)
[__exit__\(\) \(uactor.Actor method\), 11](#)
[__exit__\(\) \(uactor.ActorProxy method\), 10](#)
[_exposed_ \(uactor.Actor attribute\), 11](#)
[_method_to_typeid_ \(uactor.Actor attribute\), 11](#)
[_options_ \(uactor.Actor attribute\), 11](#)
[_proxies_ \(uactor.Actor attribute\), 11](#)

A

[Actor \(class in uactor\), 11](#)
[ActorManager \(class in uactor\), 10](#)
[ActorProxy \(class in uactor\), 10](#)
[AuthkeyError, 9](#)

B

[BaseProxy \(class in uactor\), 9](#)

C

[connect\(\) \(uactor.Actor class method\), 12](#)
[connect\(\) \(uactor.ActorManager method\), 10](#)
[connection_address\(\) \(uactor.ActorProxy property\), 10](#)

D

[DEFAULT_SERIALIZER \(in module uactor\), 9](#)

M

[manager_class \(uactor.Actor attribute\), 11](#)
[module](#)
 [uactor, 9](#)

P

[process\(\) \(uactor.ActorManager property\), 10](#)
[proxy\(\) \(in module uactor\), 12](#)
[proxy_class \(uactor.Actor attribute\), 11](#)
[ProxyError, 9](#)

S

[shutdown\(\) \(uactor.Actor method\), 11](#)

[shutdown\(\) \(uactor.ActorProxy method\), 10](#)
[start\(\) \(uactor.ActorManager method\), 10](#)

T

[typeid\(\) \(in module uactor\), 12](#)
[typeids\(\) \(uactor.ActorManager class method\), 10](#)

U

[uactor](#)
 [module, 9](#)
[UActorException, 9](#)