tractor

Release 0.0.0a0.dev0

Tyler Goodlet

Jun 15, 2021

CONTENTS:

A structured concurrent, async-native "actor model" built on trio and multiprocessing.

`tractor` is an attempt to bring trionic structured concurrency to distributed multi-core Python; it aims to be the Python multi-processing framework *you always wanted*.

`tractor` lets you spawn `trio` *"actors"*: processes which each run a `trio` scheduled task tree (also known as an async sandwich). *Actors* communicate by exchanging asynchronous messages and avoid sharing any state. This model allows for highly distributed software architecture which works just as well on multiple cores as it does over many hosts.

The first step to grok `tractor` is to get the basics of `trio` down. A great place to start is the trio docs and this blog post.

CHAPTER ONE

INSTALL

No PyPi release yet!

```
pip install git+git://github.com/goodboy/tractor.git
```

CHAPTER TWO

FEEL LIKE SAYING HI?

This project is very much coupled to the ongoing development of `trio` (i.e. `tractor` gets all its ideas from that brilliant community). If you want to help, have suggestions or just want to say hi, please feel free to ping me on the trio gitter channel!

CHAPTER

THREE

PHILOSOPHY

Our tenets non-comprehensively include:

- strict adherence to the concept-in-progress of *structured concurrency*
- no spawning of processes *willy-nilly*; causality is paramount!
- (remote) errors always propagate back to the parent supervisor
- verbatim support for `trio`'s cancellation system
- shared nothing architecture
- no use of *proxy* objects or shared references between processes
- an immersive debugging experience
- anti-fragility through chaos engineering

`tractor` is an actor-model-*like* system in the sense that it adheres to the 3 axioms but does not (yet) fulfil all "unrequirements" in practise. It is an experiment in applying structured concurrency constraints on a parallel processing system where multiple Python processes exist over many hosts but no process can outlive its parent. In *erlang* parlance, it is an architecture where every process has a mandatory supervisor enforced by the type system. The API design is almost exclusively inspired by trio's concepts and primitives (though we often lag a little). As a distributed computing system *tractor* attempts to place sophistication at the correct layer such that concurrency primitives are powerful yet simple, making it easy to build complex systems (you can build a "worker pool" architecture but it's definitely not required). There is first class support for inter-actor streaming using async generators and ongoing work toward a functional reactive style for IPC.

Warning: `tractor` is in alpha-alpha and is expected to change rapidly! Expect nothing to be set in stone. Your ideas about where it should go are greatly appreciated!

CHAPTER

FOUR

EXAMPLES

Note, if you are on Windows please be sure to see the *gotchas* section before trying these.

4.1 A trynamic first scene

Let's direct a couple *actors* and have them run their lines for the hip new film we're shooting:

```
import trio
import tractor

_this_module = __name__
the_line = 'Hi my name is {}'


tractor.log.get_console_log("INFO")


async def hi():
    return the_line.format(tractor.current_actor().name)


async def say_hello(other_actor):
    async with tractor.wait_for_actor(other_actor) as portal:
        return await portal.run(hi)


async def main():
    """Main tractor entry point, the "master" process (for now
    acts as the "director").
    """
    async with tractor.open_nursery() as n:
        print("Alright... Action!")

        donny = await n.run_in_actor(
            say_hello,
            name='donny',
            # arguments are always named
            other_actor='gretchen',
        )
        gretchen = await n.run_in_actor(
            say_hello,
            name='gretchen',
            other_actor='donny',
```

(continues on next page)

(continued from previous page)

```
        )
        print(await gretchen.result())
        print(await donny.result())
        print("CUTTTT CUUTT CUT!!! Donny!! You're supposed to say...")


if __name__ == '__main__':
    trio.run(main)
```

We spawn two *actors*, *donny* and *gretchen*. Each actor starts up and executes their *main task* defined by an async function, `say_hello()`. The function instructs each actor to find their partner and say hello by calling their partner's `hi()` function using something called a *portal*. Each actor receives a response and relays that back to the parent actor (in this case our "director" executing `main()`).

4.2 Actor spawning and causality

`tractor` tries to take `trio`'s concept of causal task lifetimes to multi-process land. Accordingly, `tractor`'s *actor nursery* behaves similar to `trio`'s nursery. That is, `tractor.open_nursery()` opens an `ActorNursery` which **must** wait on spawned *actors* to complete (or error) in the same causal way `trio` waits on spawned subtasks. This includes errors from any one actor causing all other actors spawned by the same nursery to be cancelled.

To spawn an actor and run a function in it, open a *nursery block* and use the `run_in_actor()` method:

```
import trio
import tractor


async def cellar_door():
    assert not tractor.is_root_process()
    return "Dang that's beautiful"


async def main():
    """The main ``tractor`` routine.
    """
    async with tractor.open_nursery() as n:

        portal = await n.run_in_actor(
            cellar_door,
            name='some_linguist',
        )

    # The ``async with`` will unblock here since the 'some_linguist'
    # actor has completed its main task ``cellar_door``.

    print(await portal.result())


if __name__ == '__main__':
    trio.run(main)
```

What's going on?

- an initial *actor* is started with `trio.run()` and told to execute its main task: `main()`

- inside `main()` an actor is *spawned* using an `ActorNusery` and is told to run a single function: `cellar_door()`
- a `portal` instance (we'll get to what it is shortly) returned from `nursery.run_in_actor()` is used to communicate with the newly spawned *sub-actor*
- the second actor, *some_linguist*, in a new *process* running a new `trio` task then executes `cellar_door()` and returns its result over a *channel* back to the parent actor
- the parent actor retrieves the subactor's *final result* using `portal.result()` much like you'd expect from a future.

This `run_in_actor()` API should look very familiar to users of `asyncio`'s run_in_executor() which uses a `concurrent.futures` Executor.

Since you might also want to spawn long running *worker* or *daemon* actors, each actor's *lifetime* can be determined based on the spawn method:

- if the actor is spawned using `run_in_actor()` it terminates when its *main* task completes (i.e. when the (async) function submitted to it *returns*). The `with tractor.open_nursery()` exits only once all actors' main function/task complete (just like the nursery in `trio`)
- actors can be spawned to *live forever* using the `start_actor()` method and act like an RPC daemon that runs indefinitely (the `with tractor.open_nursery()` won't exit) until cancelled

Here is a similar example using the latter method:

```python
import trio
import tractor


async def movie_theatre_question():
    """A question asked in a dark theatre, in a tangent
    (errr, I mean different) process.
    """
    return 'have you ever seen a portal?'


async def main():
    """The main ``tractor`` routine.
    """
    async with tractor.open_nursery() as n:

        portal = await n.start_actor(
            'frank',
            # enable the actor to run funcs from this current module
            enable_modules=[__name__],
        )

        print(await portal.run(movie_theatre_question))
        # call the subactor a 2nd time
        print(await portal.run(movie_theatre_question))

        # the async with will block here indefinitely waiting
        # for our actor "frank" to complete, but since it's an
        # "outlive_main" actor it will never end until cancelled
        await portal.cancel_actor()
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    trio.run(main)
```

The `enable_modules` *kwarg* above is a list of module path strings that will be loaded and made accessible for execution in the remote actor through a call to `Portal.run()`. For now this is a simple mechanism to restrict the functionality of the remote (and possibly daemonized) actor and uses Python's module system to limit the allowed remote function namespace(s).

`tractor` is opinionated about the underlying threading model used for each *actor*. Since Python has a GIL and an actor model by definition shares no state between actors, it fits naturally to use a multiprocessing `Process`. This allows `tractor` programs to leverage not only multi-core hardware but also distribute over many hardware hosts (each *actor* can talk to all others with ease over standard network protocols).

4.3 Cancellation

`tractor` supports `trio`'s cancellation system verbatim. Cancelling a nursery block cancels all actors spawned by it. Eventually `tractor` plans to support different supervision strategies like `erlang`.

4.4 Remote error propagation

Any task invoked in a remote actor should ship any error(s) back to the calling actor where it is raised and expected to be dealt with. This way remote actors are never cancelled unless explicitly asked or there's a bug in `tractor` itself.

```
import trio
import tractor


async def assert_err():
    assert 0


async def main():
    async with tractor.open_nursery() as n:
        real_actors = []
        for i in range(3):
            real_actors.append(await n.start_actor(
                f'actor_{i}',
                enable_modules=[__name__],
            ))

        # start one actor that will fail immediately
        await n.run_in_actor(assert_err)

    # should error here with a ``RemoteActorError`` containing
    # an ``AssertionError`` and all the other actors have been cancelled


if __name__ == '__main__':
    try:
        # also raises
        trio.run(main)
```

(continues on next page)

(continued from previous page)

```
    except tractor.RemoteActorError:
        print("Look Maa that actor failed hard, hehhh!")
```

You'll notice the nursery cancellation conducts a *one-cancels-all* supervisory strategy exactly like trio. The plan is to add more erlang strategies in the near future by allowing nurseries to accept a `Supervisor` type.

4.5 IPC using *portals*

`tractor` introduces the concept of a *portal* which is an API borrowed from `trio`. A portal may seem similar to the idea of a RPC future except a *portal* allows invoking remote *async* functions and generators and intermittently blocking to receive responses. This allows for fully async-native IPC between actors.

When you invoke another actor's routines using a *portal* it looks as though it was called locally in the current actor. So when you see a call to `await portal.run()` what you get back is what you'd expect to if you'd called the function directly in-process. This approach avoids the need to add any special RPC *proxy* objects to the library by instead just relying on the built-in (async) function calling semantics and protocols of Python.

Depending on the function type `Portal.run()` tries to correctly interface exactly like a local version of the remote built-in Python *function type*. Currently async functions, generators, and regular functions are supported. Inspiration for this API comes remote function execution but without the client code being concerned about the underlying channels system or shipping code over the network.

This *portal* approach turns out to be paricularly exciting with the introduction of asynchronous generators in Python 3.6! It means that actors can compose nicely in a data streaming pipeline.

4.6 Streaming

By now you've figured out that `tractor` lets you spawn process based *actors* that can invoke cross-process (async) functions and all with structured concurrency built in. But the **real cool stuff** is the native support for cross-process *streaming*.

4.6.1 Asynchronous generators

The default streaming function is simply an async generator definition. Every value *yielded* from the generator is delivered to the calling portal exactly like if you had invoked the function in-process meaning you can `async for` to receive each value on the calling side.

As an example here's a parent actor that streams for 1 second from a spawned subactor:

```
from itertools import repeat
import trio
import tractor

tractor.log.get_console_log("INFO")


async def stream_forever():
    for i in repeat("I can see these little future bubble things"):
        # each yielded value is sent over the ``Channel`` to the
        # parent actor
        yield i
```

(continues on next page)

(continued from previous page)

```python
        await trio.sleep(0.01)


async def main():

    # stream for at most 1 seconds
    with trio.move_on_after(1) as cancel_scope:

        async with tractor.open_nursery() as n:

            portal = await n.start_actor(
                'donny',
                enable_modules=[__name__],
            )

            # this async for loop streams values from the above
            # async generator running in a separate process
            async with portal.open_stream_from(stream_forever) as stream:
                async for letter in stream:
                    print(letter)

    # we support trio's cancellation system
    assert cancel_scope.cancelled_caught
    assert n.cancelled


if __name__ == '__main__':
    trio.run(main)
```

By default async generator functions are treated as inter-actor *streams* when invoked via a portal (how else could you really interface with them anyway) so no special syntax to denote the streaming *service* is necessary.

4.6.2 Channels and Contexts

If you aren't fond of having to write an async generator to stream data between actors (or need something more flexible) you can instead use a `Context`. A context wraps an actor-local spawned task and a `Channel` so that tasks executing across multiple processes can stream data to one another using a low level, request oriented API.

A `Channel` wraps an underlying *transport* and *interchange* format to enable *inter-actor-communication*. In its present state `tractor` uses TCP and msgpack.

As an example if you wanted to create a streaming server without writing an async generator that *yields* values you instead define a decorated async function:

```python
@tractor.stream
async def streamer(ctx: tractor.Context, rate: int = 2) -> None:
   """A simple web response streaming server.
   """
   while True:
      val = await web_request('http://data.feed.com')

      # this is the same as ``yield`` in the async gen case
      await ctx.send_yield(val)

      await trio.sleep(1 / rate)
```

You must decorate the function with `@tractor.stream` and declare a `ctx` argument as the first in your function signature and then `tractor` will treat the async function like an async generator - as a stream from the calling/client side.

This turns out to be handy particularly if you have multiple tasks pushing responses concurrently:

```
async def streamer(
   ctx: tractor.Context,
   rate: int = 2
) -> None:
   """A simple web response streaming server.
   """
   while True:
      val = await web_request(url)

      # this is the same as ``yield`` in the async gen case
      await ctx.send_yield(val)

      await trio.sleep(1 / rate)


@tractor.stream
async def stream_multiple_sources(
   ctx: tractor.Context,
   sources: List[str]
) -> None:
   async with trio.open_nursery() as n:
      for url in sources:
         n.start_soon(streamer, ctx, url)
```

The context notion comes from the context in nanomsg.

4.6.3 A full fledged streaming service

Alright, let's get fancy.

Say you wanted to spawn two actors which each pull data feeds from two different sources (and wanted this work spread across 2 cpus). You also want to aggregate these feeds, do some processing on them and then deliver the final result stream to a client (or in this case parent) actor and print the results to your screen:

```
import time
import trio
import tractor


# this is the first 2 actors, streamer_1 and streamer_2
async def stream_data(seed):
    for i in range(seed):
        yield i
        await trio.sleep(0)  # trigger scheduler


# this is the third actor; the aggregator
async def aggregate(seed):
    """Ensure that the two streams we receive match but only stream
    a single set of values to the parent.
    """
```

(continues on next page)

(continued from previous page)

```
    async with tractor.open_nursery() as nursery:
        portals = []
        for i in range(1, 3):
            # fork point
            portal = await nursery.start_actor(
                name=f'streamer_{i}',
                enable_modules=[__name__],
            )

            portals.append(portal)

        send_chan, recv_chan = trio.open_memory_channel(500)

        async def push_to_chan(portal, send_chan):

            # TODO: https://github.com/goodboy/tractor/issues/207
            async with send_chan:
                async with portal.open_stream_from(stream_data, seed=seed) as stream:
                    async for value in stream:
                        # leverage trio's built-in backpressure
                        await send_chan.send(value)

            print(f"FINISHED ITERATING {portal.channel.uid}")

        # spawn 2 trio tasks to collect streams and push to a local queue
        async with trio.open_nursery() as n:

            for portal in portals:
                n.start_soon(push_to_chan, portal, send_chan.clone())

            # close this local task's reference to send side
            await send_chan.aclose()

            unique_vals = set()
            async with recv_chan:
                async for value in recv_chan:
                    if value not in unique_vals:
                        unique_vals.add(value)
                        # yield upwards to the spawning parent actor
                        yield value

                assert value in unique_vals

            print("FINISHED ITERATING in aggregator")

        await nursery.cancel()
        print("WAITING on `ActorNursery` to finish")
    print("AGGREGATOR COMPLETE!")


# this is the main actor and *arbiter*
async def main():
    # a nursery which spawns "actors"
    async with tractor.open_nursery(
        arbiter_addr=('127.0.0.1', 1616)
    ) as nursery:
```

(continues on next page)

(continued from previous page)

```
        seed = int(1e3)
        pre_start = time.time()

        portal = await nursery.start_actor(
            name='aggregator',
            enable_modules=[__name__],
        )

        async with portal.open_stream_from(
            aggregate,
            seed=seed,
        ) as stream:

            start = time.time()
            # the portal call returns exactly what you'd expect
            # as if the remote "aggregate" function was called locally
            result_stream = []
            async for value in stream:
                result_stream.append(value)

        await portal.cancel_actor()

        print(f"STREAM TIME = {time.time() - start}")
        print(f"STREAM + SPAWN TIME = {time.time() - pre_start}")
        assert result_stream == list(range(seed))
        return result_stream


if __name__ == '__main__':
    final_stream = trio.run(main)
```

Here there's four actors running in separate processes (using all the cores on you machine). Two are streaming by *yielding* values from the `stream_data()` async generator, one is aggregating values from those two in `aggregate()` (also an async generator) and shipping the single stream of unique values up the parent actor (the `'MainProcess'` as `multiprocessing` calls it) which is running `main()`.

4.7 Actor local (aka *process global*) variables

Although `tractor` uses a *shared-nothing* architecture between processes you can of course share state between tasks running *within* an actor (since a *trio.run()* runtime is single threaded). `trio` tasks spawned via multiple RPC calls to an actor can modify *process-global-state* defined using Python module attributes:

```
# a per process cache
_actor_cache: Dict[str, bool] = {}


def ping_endpoints(endpoints: List[str]):
    """Start a polling process which runs completely separate
    from our root actor/process.

    """

    # This runs in a new process so no changes # will propagate
    # back to the parent actor
```

(continues on next page)

(continued from previous page)

```
    while True:

        for ep in endpoints:
            status = await check_endpoint_is_up(ep)
            _actor_cache[ep] = status

        await trio.sleep(0.5)


async def get_alive_endpoints():

    nonlocal _actor_cache

    return {key for key, value in _actor_cache.items() if value}


async def main():

    async with tractor.open_nursery() as n:

        portal = await n.run_in_actor(ping_endpoints)

        # print the alive endpoints after 3 seconds
        await trio.sleep(3)

        # this is submitted to be run in our "ping_endpoints" actor
        print(await portal.run(get_alive_endpoints))
```

You can pass any kind of (*msgpack*) serializable data between actors using function call semantics but building out a state sharing system per-actor is totally up to you.

4.8 Service Discovery

Though it will be built out much more in the near future, `tractor` currently keeps track of actors by `(name: str, id: str)` using a special actor called the *arbiter*. Currently the *arbiter* must exist on a host (or it will be created if one can't be found) and keeps a simple `dict` of actor names to sockets for discovery by other actors. Obviously this can be made more sophisticated (help me with it!) but for now it does the trick.

To find the arbiter from the current actor use the `get_arbiter()` function and to find an actor's socket address by name use the `find_actor()` function:

```
import trio
import tractor

tractor.log.get_console_log("INFO")


async def main(service_name):

    async with tractor.open_nursery() as an:
        await an.start_actor(service_name)

        async with tractor.get_arbiter('127.0.0.1', 1616) as portal:
            print(f"Arbiter is listening on {portal.channel}")
```

(continues on next page)

(continued from previous page)

```
        async with tractor.wait_for_actor(service_name) as sockaddr:
            print(f"my_service is found at {sockaddr}")

        await an.cancel()


if __name__ == '__main__':
    trio.run(main, 'some_actor_name')
```

The `name` value you should pass to `find_actor()` is the one you passed as the *first* argument to either `trio.run()` or `ActorNursery.start_actor()`.

4.9 Running actors standalone

You don't have to spawn any actors using `open_nursery()` if you just want to run a single actor that connects to an existing cluster. All the comms and arbiter registration stuff still works. This can somtimes turn out being handy when debugging mult-process apps when you need to hop into a debugger. You just need to pass the existing *arbiter*'s socket address you'd like to connect to:

```
import trio
import tractor

async def main():

    async with tractor.open_root_actor(
        arbiter_addr=('192.168.0.10', 1616)
    ):
        await trio.sleep_forever()

trio.run(main)
```

4.10 Choosing a process spawning backend

`tractor` is architected to support multiple actor (sub-process) spawning backends. Specific defaults are chosen based on your system but you can also explicitly select a backend of choice at startup via a `start_method` kwarg to `tractor.open_nursery()`.

Currently the options available are:

- `trio`: a `trio`-native spawner which is an async wrapper around `subprocess`
- `spawn`: one of the stdlib's `multiprocessing` start methods
- `forkserver`: a faster `multiprocessing` variant that is Unix only

4.10.1 `trio`

The `trio` backend offers a lightweight async wrapper around `subprocess` from the standard library and takes advantage of the `trio.` open_process API.

4.10.2 `multiprocessing`

There is support for the stdlib's `multiprocessing` start methods. Note that on Windows *spawn* it the only supported method and on *nix systems *forkserver* is the best method for speed but has the caveat that it will break easily (hangs due to broken pipes) if spawning actors using nested nurseries.

In general, the `multiprocessing` backend **has not proven reliable** for handling errors from actors more then 2 nurseries *deep* (see #89). If you for some reason need this consider sticking with alternative backends.

Windows "gotchas"

On Windows (which requires the use of the stdlib's *multiprocessing* package) there are some gotchas. Namely, the need for calling freeze_support() inside the `__main__` context. Additionally you may need place you *tractor* program entry point in a seperate *__main__.py* module in your package in order to avoid an error like the following

```
Traceback (most recent call last):
  File "C:\ProgramData\Miniconda3\envs\tractor19030601\lib\site-packages\tractor\_
↪actor.py", line 234, in _get_rpc_func
    return getattr(self._mods[ns], funcname)
KeyError: '__mp_main__'
```

To avoid this, the following is the **only code** that should be in your main python module of the program:

```
# application/__main__.py
import trio
import tractor
import multiprocessing
from . import tractor_app

if __name__ == '__main__':
    multiprocessing.freeze_support()
    trio.run(tractor_app.main)
```

And execute as:

```
python -m application
```

As an example we use the following code to test all documented examples in the test suite on windows:

```
"""
Needed on Windows.

This module is needed as the program entry point for invocation
with ``python -m <modulename>``. See the solution from @chrizzFTD
here:

    https://github.com/goodboy/tractor/pull/61#issuecomment-470053512

"""
if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    import multiprocessing
    multiprocessing.freeze_support()
    # ``tests/test_docs_examples.py::test_example`` will copy each
    # script from this examples directory into a module in a new
    # temporary dir and name it test_example.py. We import that script
    # module here and invoke it's ``main()``.
    from . import test_example
    test_example.trio.run(test_example.main)
```

See #61 and #79 for further details.

4.11 Enabling logging

Considering how complicated distributed software can become it helps to know what exactly it's doing (even at the lowest levels). Luckily `tractor` has tons of logging throughout the core. `tractor` isn't opinionated on how you use this information and users are expected to consume log messages in whichever way is appropriate for the system at hand. That being said, when hacking on `tractor` there is a prettified console formatter which you can enable to see what the heck is going on. Just put the following somewhere in your code:

```
from tractor.log import get_console_log
log = get_console_log('trace')
```

CHAPTER

FIVE

WHAT THE FUTURE HOLDS

Stuff I'd like to see `tractor` do real soon:

- TLS, duh.
- erlang-like supervisors
- native support for nanomsg as a channel transport
- native gossip protocol support for service discovery and arbiter election
- a distributed log ledger for tracking cluster behaviour
- a slick multi-process aware debugger much like in celery but with better pdb++ support
- an extensive chaos engineering test suite
- support for reactive programming primitives and native support for asyncitertools like libs
- introduction of a capability-based security model