async_stagger Documentation

twisteroid ambassador

Oct 20, 2020

CONTENTS OF THE DOCUMENTATION:

Project home page: https://github.com/twisteroidambassador/async_stagger

Check out the *project's README file* for the elevator pitch.

CHAPTER

ONE

ASYNC_STAGGER: HAPPY EYEBALLS IN `ASYNCIO`

1.1 Quick, what's the situation?

To get all the benefits of Happy Eyeballs connection establishment algorithm, simply use `async_stagger.open_connection` like you would use `asyncio.open_connection`:

```
reader, writer = await async_stagger.open_connection('www.example.com', 80)
```

Now your connections are more dual-stack friendly and will complete faster! A replacement for `loop.create_connection` is also provided.

1.2 The long version

1.2.1 What is Happy Eyeballs, and why should I use it?

Happy Eyeballs is an algorithm for establishing TCP connections to destinations specified by host names. It is described in **RFC 6555** and **RFC 8305**. The primary benefit is that when host name resolution returns multiple addresses, and some of the address are unreachable, Happy Eyeballs will establish the connection much faster than conventional algorithms. For more information, check the Wikipedia article on Happy Eyeballs.

Python's standard library provides several high-level methods of establishing TCP connections towards a host name: The **socket** module has `socket.create_connection`, and **asyncio** has `loop.create_connection` and `asyncio.open_connection`. By default, these methods have the same behavior when a host name resolves to several IP addresses: they try to connect to the first address in the list, and only after the attempt fails (which may take tens of seconds) will the second one be tried, and so on. In contrast, the Happy Eyeballs algorithm will start an attempt with the second IP address in parallel to the first one hasn't completed after some time, typically around 300 milliseconds. As a result several attempts may be in flight at the same time, and whenever one of the attempts succeed, all other connections are cancelled, and the winning connection is used. This means a much shorter wait before one of the IP addresses connect successfully.

Happy Eyeballs is particularly important for dual-stack clients, when some hosts may have resolvable IPv6 addresses that are somehow unreachable.

Starting from Python 3.8, stock **asyncio** also supports Happy Eyeballs. See below for a comparison.

1.2.2 What does `async_stagger` has to offer?

`async_stagger` provides `open_connection` and `create_connection` with Happy Eyeballs support. They are mostly drop-in replacements for their `asyncio` counterparts, and support most existing arguments. (There are small differences: `create_connection` takes a *loop* argument instead of being a method on an event loop. Also, these two methods do not support the *sock* argument.) Another public coroutine `create_connected_sock` returns a connected `socket.socket` object. Check the documentation for details.

These methods implements many features specified in **RFC 8305** Happy Eyeballs v2, which extends and obsoletes **RFC 6555**. In particular, asynchronous address resolution, destination address interleaving by family and staggered connection attempts are implemented.

1.2.3 Happy Eyeballs sounds great! I want to use similar logic somewhere else!

You're in luck! `async_stagger` actually exposes the underlying scheduling logic as a reusable block: `staggered_race`. It can be use when:

- There are several ways to achieve one goal. Some of the ways may fail, but you have to try it to find out.
- Making attempts strictly in sequence is too slow.
- You want to parallelize, but also don't want to start the attempts all at the same time. Maybe you want to give preference to some of the attempts, so they should be started earlier and given more time to complete. Maybe you want to avoid straining the system with simultaneous attempts.
- An attempt done half-way can be rolled back safely.

1.2.4 Where can I get it?

`async_stagger` requires Python 3.6 or later. (v0.2.0 onwards uses more new features in 3.6 such as async generators and async comprehensions, so it will probably require more than cosmetic changes to make it run on 3.5.) It does not have any external dependencies. Install it from PyPI the usual way:

```
pip install async-stagger
```

The documentation can be found here: http://async-stagger.readthedocs.io/en/latest/

This project is under active development, and APIs may change in the future. Check out the Changelog in the documentation.

This project is licensed under the MIT license.

1.3 Python 3.8 Has Native Happy Eyeballs Now

I contributed an implementation of Happy Eyeballs to upstream **asyncio**, and it landed in Python 3.8: see the docs for details.

That implementation is essentially an early version of this package, so it lacks these features:

- Async address resolution
- Detailed exception report
- The `local_addrs` argument (as opposed to `local_addr`)

Still, it should be sufficient for most scenarios, and it's right there in the standard library.

1.4 Miscellaneous Remarks

Asynchronous address resolution is added in v0.2.1. With that, I feel that the package should be fairly feature-complete.

I have implemented Happy Eyeballs-like algorithms in some of my other projects, and this module reflects the things I have learned. However I have yet to eat my own dog food and actually import this module from those other projects. I would love to hear people's experience using this module in real world conditions.

bpo-31861 talks about adding native `aiter` and `anext` functions either as builtins or to the `operator` module. Well, I want them NAO!!!one!!!eleventy!! So I borrowed the implementations from that bpo and put them in the `aitertools` submodule. I have only kept the one-argument forms; In particular, the two-argument `iter` function is so disparate from the one-argument version, that I don't think they belong to the same function at all, and there really shouldn't be a need for `aiter` to emulate that behavior.

1.5 Acknowledgments

The Happy Eyeballs scheduling algorithm implementation is inspired by the implementation in trio.

CHAPTER

TWO

ASYNC_STAGGER API REFERENCE

2.1 The Main Package

await async_stagger.**create_connected_sock**(*host*, *port*, ***, *family=<AddressFamily.AF_UNSPEC: 0>*, *proto=0*, *flags=0*, *local_addr=None*, *local_addrs=None*, *delay=0.25*, *interleave=1*, *async_dns=False*, *resolution_delay=0.05*, *detailed_exceptions=False*, *loop=None*)

Connect to *(host, port)* and return a connected socket.

This function implements **RFC 6555** Happy Eyeballs and some features of **RFC 8305** Happy Eyeballs v2. When a host name resolves to multiple IP addresses, connection attempts are made in parallel with staggered start times, and the one completing fastest is used. The resolved addresses can be interleaved by address family, so even if network connectivity for one address family is broken (when IPv6 fails, for example), connections still complete quickly. IPv6 and IPv4 addresses of a hostname can also be resolved in parallel.

(Some fancier features specified in **RFC 8305**, like statefulness and features related to NAT64 and DNS64 are not implemented. Destination address sorting is left for the operating system; it is assumed that the addresses returned by `getaddrinfo()` is already sorted according to OS's preferences.)

Most of the arguments should be familiar from the various `socket` and `asyncio` methods. *delay*, *interleave*, *async_dns* and *resolution_delay* control Happy Eyeballs-specific behavior. *local_addrs* is a new argument providing new features not specific to Happy Eyeballs.

Parameters

- **host** (`Union[str, bytes, None]`) – Host name to connect to. Unlike `asyncio.create_connection()` there is no default, but it's still possible to manually specify *None* here.
- **port** (`Union[str, bytes, int, None]`) – Port number to connect to. Similar to **host**, *None* can be specified here as well.
- **family** (`int`) – Address family. Specify `socket.AF_INET` or `socket.AF_INET6` here to limit the type of addresses used. See documentation on the `socket` module for details.
- **proto** (`int`) – Socket protocol. Since the socket type is always `socket.SOCK_STREAM`, proto can usually be left unspecified.
- **flags** (`int`) – Flags passed to `getaddrinfo()`. See documentation on `socket.getaddrinfo()` for details.

- **`local_addr`** (`Optional[Tuple]`) – *(local_host, local_port)* tuple used to bind the socket to locally. The *local_host* and *local_port* are looked up using `getaddrinfo()` if necessary, similar to *host* and *port*.
- **`local_addrs`** (`Optional[Iterable[Tuple]]`) – An iterable of (local_host, local_port) tuples, all of which are candidates for locally binding the socket to. This allows e.g. providing one IPv4 and one IPv6 address. Addresses are looked up using `getaddrinfo()` if necessary.
- **`delay`** (`Optional[float]`) – Amount of time to wait before making connections to different addresses. This is the "Connect Attempt Delay" as defined in **RFC 8305**.
- **`interleave`** (`int`) – Whether to interleave resolved addresses by address family. 0 means not to interleave and simply use the returned order. An integer >= 1 is interpreted as "First Address Family Count" defined in **RFC 8305**, i.e. the reordered list will have this many addresses for the first address family, and the rest will be interleaved one to one.
- **`async_dns`** (`bool`) – Do asynchronous DNS resolution, where IPv6 and IPv4 addresses are resolved in parallel, and connection attempts can be made as soon as either address family is resolved. This behavior is described in **RFC 8305#section-3**.
- **`resolution_delay`** (`float`) – Amount of time to wait for IPv6 addresses to resolve if IPv4 addresses are resolved first. This is the "Resolution Delay" as defined in **RFC 8305**.
- **`detailed_exceptions`** (`bool`) – Determines what exception to raise when all connection attempts fail. If set to True, an instance of *`HappyEyeballsConnectError`* is raised, which contains the individual exceptions raised by each connection and address resolution attempt. When set to false (default), an exception is raised the same way as `asyncio.create_connection()`: if all the connection attempt exceptions have the same `str`, one of them is raised, otherwise an instance of *OSError* is raised whose message contains `str` representations of all connection attempt exceptions.
- **`loop`** (`Optional[AbstractEventLoop]`) – Event loop to use.

Return type `socket`

Returns The connected `socket.socket` object.

New in version v0.1.3: the *local_addrs* parameter.

New in version v0.2.1: the *async_dns* and *resolution_delay* parameters.

`await` `async_stagger.`**`create_connection`**(*protocol_factory*, *host*, *port*, ***, *loop=None*, ***kwargs*)

Connect to *(host, port)* and return *(transport, protocol)*.

This function does the same thing as `asyncio.AbstractEventLoop.create_connection()`, only more awesome with Happy Eyeballs. Refer to that function's documentation for explanations of these arguments: *protocol_factory*, *ssl*, and *server_hostname*. Refer to *`create_connected_sock()`* for all other arguments.

Return type `Tuple[Transport, Protocol]`

Returns *(transport, protocol)*, the same as `asyncio.AbstractEventLoop.create_connection()`.

`await` `async_stagger.`**`open_connection`**(*host*, *port*, ***, *loop=None*, ***kwargs*)

Connect to (host, port) and return (reader, writer).

This function does the same thing as `asyncio.open_connection()`, with added awesomeness of Happy Eyeballs. Refer to the documentation of that function for what *limit* does, and refer to *`create_connection()`* and *`create_connected_sock()`* for everything else.

Return type `Tuple[StreamReader, StreamWriter]`

Returns *(reader, writer)*, the same as `asyncio.open_connection()`.

await async_stagger.**staggered_race**(*coro_fns*, *delay*, ***, *loop=None*)

Run coroutines with staggered start times and take the first to finish.

This function takes an async iterable of coroutine functions. The first one is retrieved and started immediately. From then on, whenever the immediately preceding one fails (raises an exception), or when *delay* seconds has passed, the next coroutine is retrieved and started. This continues until one of the coroutines complete successfully, in which case all others are cancelled, or until all coroutines fail.

The coroutines provided should be well-behaved in the following way:

- They should only `return` if completed successfully.
- They should always raise an exception if they did not complete successfully. In particular, if they handle cancellation, they should probably reraise, like this:

```
try:
    # do work
except asyncio.CancelledError:
    # undo partially completed work
    raise
```

Parameters

- **coro_fns** (`AsyncIterable[Callable[[], Awaitable]]`) – an async iterable of coroutine functions, i.e. callables that return a coroutine object when called. Use `functools.partial()` or lambdas to pass arguments. If you want to use a regular iterable here, wrap it with *`aiter_from_iter()`*.
- **delay** (`Optional[float]`) – amount of time, in seconds, between starting coroutines. If `None`, the coroutines will run sequentially.
- **loop** (`Optional[AbstractEventLoop]`) – the event loop to use.

Return type `Tuple[Any, Optional[int], List[Optional[Exception]], Optional[Exception]]`

Returns

tuple *(winner_result, winner_index, coro_exc, aiter_exc)* where

- *winner_result*: the result of the winning coroutine, or `None` if no coroutines won.
- *winner_index*: the index of the winning coroutine in `coro_fns`, or `None` if no coroutines won. If the winning coroutine may return None on success, *winner_index* can be used to definitively determine whether any coroutine won.
- *coro_exc*: list of exceptions raised by the coroutines. `len(exceptions)` is equal to the number of coroutines actually started, and the order is the same as in `coro_fns`. The winning coroutine's entry is `None`.
- *aiter_exc*: exception raised by the *coro_fns* async iterable, or `None` if *coro_fns* was iterated to completion without raising any exception.

Changed in version v0.2.0: *coro_fns* argument now takes an async iterable instead of a regular iterable.

Changed in version v0.3.0: The return value is now a 4-tuple. *aiter_exc* is added.

2.2 aitertools

Tools for working with async iterators.

`async_stagger.aitertools.`**`aiter`**`(`*aiterable*`)`

Return an async iterator from an async iterable.

If an `aiter` function is available as a builtin or in the `operator` module, it is imported into *`async_stagger.aitertools`*, and this function will not be defined. Only when a stock `aiter` is not available will this function be defined.

Unlike the built-in `iter()`, this only support one argument, and does not support the two-argument (callable, sentinel) usage.

Adapted from implementation attached to https://bugs.python.org/issue31861 by Davide Rizzo.

Parameters **`aiterable`** (`AsyncIterable`[~T]) – The async iterable.

Return type `AsyncIterator`[~T]

Returns The async iterator produced from the given async iterable.

`async for ... in` `async_stagger.aitertools.`**`aiter_from_iter`**`(`*iterable*`)`

Wrap an async iterator around a regular iterator.

Parameters **`iterable`** (`Iterable`[~T]) – a regular iterable.

Return type `AsyncIterator`[~T]

Returns An async iterator yielding the same items as the original iterable.

`await` `async_stagger.aitertools.`**`aiterclose`**`(`*aiterator*`)`

Close the async iterator if possible.

Async generators have an `aclose()` method that closes the generator and cleans up associated resources. Plain async iterators do not have anything similar, but **PEP 533** suggests adding an `__aiterclose__()` method, and having it called automatically when exiting from an `async with` loop.

This function tries to close the async iterator using either method, and if neither is available, does nothing.

Parameters **`aiterator`** (`AsyncIterator`) – the async iterator to close.

`async_stagger.aitertools.`**`anext`**`(`*aiterator*`)`

Return the next item from an async iterator.

If an `anext` function is available as a builtin or in the `operator` module, it is imported into *`async_stagger.aitertools`*, and this function will not be defined. Only when a stock `anext` is not available will this function be defined.

Unlike the built-in `next()`, this does not support providing a default value.

This is a regular function that returns an awaitable, so usually you should await its result: `await anext(it)`

Adapted from implementation attached to https://bugs.python.org/issue31861 by Davide Rizzo.

Parameters **`aiterator`** (`AsyncIterator`[~T]) – the async iterator.

Return type `Awaitable`[~T]

Returns An awaitable that will return the next item in the iterator.

`async for ... in` `async_stagger.aitertools.`**`product`**`(`**aiterables*, *repeat=1*`)`

Async version of `itertools.product()`.

Compute the cartesian product of input iterables. The arguments are analogous to its `itertools` counterpart.

The input async iterables are evaluated lazily. As a result the last input iterable is iterated and exhausted first, then the next-to-last is iterated, and so on.

Parameters

- **`aiterables`** (`AsyncIterable`) – input async iterables.
- **`repeat`** (`int`) – used to compute the product of input async iterables with themselves.

Return type `AsyncIterator`

2.3 `exceptions`

exception `async_stagger.exceptions.`**`HappyEyeballsConnectError`**

Encapsulate all exceptions encountered during connection.

This exception is raised when *`create_connected_sock()`* fails with the *detailed_exceptions* argument set. The *args* of this exception consists of a list of exceptions occurred during all connection attempts and address resolution.

CHAPTER

THREE

CHANGELOG

3.1 v0.3.1

Added support for Python 3.8.

Added `aiterclose()` that tries to close an async iterator.

3.2 v0.3.0

Backwards incompatible change: Added new return value *aiter_exc* to `staggered_race()`. It contains the exception raised by the async iterator, if any.

Added new argument *detailed_exceptions* to `create_connected_sock()`. When set to True, when the connection fails, a `HappyEyeballsConnectError` is raised, containing all the exceptions raised by the connect / resolution tasks.

Added debug logging features.

3.3 v0.2.1

Added support for asynchronous address resolution: IPv6 and IPv4 addresses for a hostname can be resolved in parallel, and connection attempts may start as soon as either address family is resolved. This reduces time needed for connection establishment in cases where resolution for a certain address family is slow.

3.4 v0.2.0

Backwards incompatible change: `staggered_race()` now takes an async iterable instead of a regular iterable for its *coro_fns* argument.

A new module `aitertools` is added, containing tools for working with async iterators. Among other things, implementations for `aiter()` and `anext()` are provided, analogous to the built-in functions `iter()` and `next()`.

Implementation detail: Code for resolving host names to IP addresses are moved to their own module and made to yield results as async iterables.

3.5 v0.1.3

Added support for multiple local addresses.

3.6 v0.1.2

Fixed several bugs.

3.7 v0.1.1

The first real release. Implements stateless Happy Eyeballs.

Contents of this page

CHAPTER

FOUR

QUICK START

4.1 Installation

Install through PyPI as usual:

```
pip install async-stagger
```

Python 3.6 or above is required.

4.2 Making TCP connections using Happy Eyeballs

To quickly get the benefit of Happy Eyeballs, simply use *async_stagger.create_connection()* and *async_stagger.open_connection()* where you would use their `asyncio` counterparts. Modifications required are minimal, since they support all the usual arguments except *sock*, and all new arguments are optional and have sane defaults.

Alternatively, use *async_stagger.create_connected_sock()* to create a connected `socket.socket` object, and use it as you wish.

4.3 Using the underlying scheduling logic

The Happy Eyeballs scheduling logic, i.e. "run coroutines with staggered start times, wait for one to complete, cancel all others", is exposed in a reusable form in *async_stagger.staggered_race()*.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX