net Documentation

Release 0.3.1

Alex Hatfield

Mar 23, 2019

Contents:

CHAPTER 1

app-net

Pure python peer-to-peer interfacing framework. Define functions that can be executed from within the running instance of python, just like a normal function. Or execute the same function on a remote peer running either the same application or a compatible function and return the result as though it was run locally.

Link to the Documentation.

1.1 Basic Example

Below is a basic example of defining an application that is running on 2 separate hosts independently. We will define a simple function that will take a positional argument and keyword argument then multiplies them together and returns the result.

First we will define our function

```
import net

@net.connect
def my_function(some_arg, some_kwarg=5):
    return some_arg * some_kwarg
```

Now we can launch 2 instances of python. It can be either on the same or remote host, net handles this through peer ids.

```
>>> import net
>>> # run this function locally on this instance of python
>>> my_function(5, some_kwarg=10)
50
>>> # get all peers on the network
>>> for peer_id in net.get_peers():
...     # execute the same function but on other instances of python and return the␣
↪results
...     print(my_function(5, some_kwarg=10, peer=peer_id))
...
50
```

CHAPTER 2

Installation

2.1 Stable release

To install net, run this command in your terminal:

```
$ pip install app-net
```

This is the preferred method to install net, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for net can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/aldmbmtl/net
```

Or download the tarball:

```
$ curl  -OL https://github.com/aldmbmtl/net/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

3.1 Core Concepts

app-net uses peer-to-peer socket servers to execute code both locally and remotely. The first thing to understand is the difference between **local** and **remote** execution of a function.

3.1.1 Local

When you launch python and you execute a function, it will execute inside that instance, obviously. app-net requires you the developer to define the peer id to execute the function on. If you don't tell the function where to execute the code, it will default to a normal pass-through. This makes development and testing easier. The response locally is expected to match a remote peers response.

3.1.2 Remote

When you execute a function, you can tell it to **connect** to a different instance of python, execute the code, and return the result through the socket response. The thing to understand is that a **remote** instance doesn't need to be on another host. Meaning, if you have 2 instances of python running app-net on the same host, they can communicate the same way they would if they were on a different host.

3.2 A Basic Example

Each connected function is registered using the functions `func.__module__` and `func.__name__` attributes and then encoded into base64 for easier transit between peers. This connection identifier is called a "tag". So, when Peer1 wants to execute a function on Peer2, it will send a JSON request that has the args, kwargs and the tag. The tag is then used to find the function in Peer2's registry and then pass the args and kwargs to that function. If it succeeds, the result is sent back to Peer1. If not, the traceback is captured and sent back and Peer1 will throw a matching error.

We are going to write a very simple application that will multiply 2 values together. Then we will flag this function as a "connect" function. Then we will launch 2 instances on our local host, and trigger execution calls between the instances.

Firstly, we will define a our basic multiply function. Then we will flag it with the `net.connect` decorator. This connect function will launch a `net.Peer` server and register our `multiply_values` function with it.

```
import net

# application code
@net.connect()
def multiply_values(val1, val2):
    return val1 * val2
```

```
>>> import net
>>>
>>> # get all net peers reachable on local host and the local area network.
>>> for peer_id in net.get_peers():
>>>     #
>>>     print(multiply_values(5, 10, peer=peer_id))
50
...
```


```
>>> import net
>>>
>>> # get all net peers reachable on local host and the local area network.
>>> for peer_id in net.get_peers():
>>>     #
>>>     print(multiply_values(5, 10, peer=peer_id))
50
...
```

CHAPTER 4

API Reference

4.1 Environment

All of the following are environment variables that can be set to configure net. Each variable is prefixed with "NET_{value}".

4.1.1 Network Thread Limit

`net.THREAD_LIMIT`

Default: 5

For larger networks, you may want to increase the thread count. By default, this is set to 5. When scanning the network for peers, the total number of hosts is load balanced between the thread count you provide in this variable.

4.1.2 Port Configuration

`net.PORT_START`

Default: 3010

This is the starting port that the peers will attempt to bind to.

`net.PORT_RANGE`

Default: 5

This is the range of ports that you want the port to try to bind to. If the default is 3010, net will scan 3010 - 3015 for a port.

4.1.3 Peer Configuration

`net.GROUP`

Default: None

You can group your peers together by defining the group it belongs to. This helps Peers find compatible peers or collections.

`net.IS_HUB`

Default: False

If you have a single peer that should be the center of an application, you can identify it through this variable. When you run `net.info` on a peer with this flag, it will return True in the hub field of the friendly_id.

4.1.4 Development Configuration

`net.DEV`

Default: None

This will activate the DEBUG level for the net logger. This helps a ton if you are having trouble tracking communication between peers.

4.2 Decorators

`net.connect`(*tag=None*)

Registers a function as a connection. This will be tagged and registered with the Peer server. The tag is a base64 encoded path to the function or can be manually tagged with the tag parameter. Tagging a named function allows you to interconnect functions between code bases.

For example, a connected function with no tag is tied to the `func.__module__` + `func.__name__`. This means the peers will only know which functions are compatible based on the namespace staying the same.

```
# app version 1 running on PeerA
app/
  module/
    function

# app version 2 running on PeerB
app/
  module/
    function2 <- # renamed from function
```

In the above example, PeerA could make a request to PeerB to execute "app.module.function". But that function no longer exists as far as PeerB is concerned. The source code and functionality could be exactly the same, but the logical location is different and therefore will fail.

```
# app version 1 running on PeerA
app/
  module/
    function (tagged: "MyTaggedFunction")

# app version 2 running on PeerB
app/
  module/
    function2 (tagged: "MyTaggedFunction")
```

In the above example, we have tagged function and function2 with the same tag, "MyTaggedFunction". Now when PeerA requests to execute, it will request that PeerB executes "MyTaggedFunction" which is attached to the new renamed function.

Standard no tagging

```
@net.connect()
def your_function(some_value):
    return some_value
```

Custom tagging

```
@net.connect("MyTaggedFunction")
def your_function(some_value):
    return some_value
```

net.**subscribe**(*event*, *peers=None*)

Subscribe to an event on another peer or set of peers. When the peer triggers an event using `net.event`, the peer will take the arguments passed and forward them to this function. By default, this will subscribe to all peers. You can also manually filter the peers by selectively passing in only the peers you want to subscribe to using the `peers` keyword argument.

Subscribe to "some_event" on group1 peers only.

```
group1_peers = net.peers(groups=['group1'])

@net.subscribe("some_event", group1_peers)
def your_function(subscription_args, subscription_kwarg=None):
    return some_value
```

Subscribe to "some_event" on a single peer.

```
peer = net.peers()[0]

@net.subscribe("some_event", peer)
def your_function(subscription_args, subscription_kwarg=None):
    return some_value
```

Subscribe to "some_event" on all peers.

```
@net.subscribe("some_event")
def your_function(subscription_args, subscription_kwarg=None):
    return some_value
```

net.**flag**(*name*)

Register a function as a flag handler for the peer server.

Parameters **name** – str

4.3 Functions

These functions are in place to help with discovering the network and interacting with other peers.

net.**peers**(*refresh=False*, *groups=None*)

Get a list of all peers on your network. This is a cached values since the call to graph the network can be long.

The initial call to this will hang for a few seconds. Under the hood, it is making a shell call to `arp -a` which will walk your network and find all hosts.

Standard call to get the peers on your network.

```
all_peers = net.peers()
```

Refresh all peers in the cache

```
all_peers = net.peers(refresh=True)
```

Refresh the cache with peers in group1

```
all_peers = net.peers("group1", refresh=True)
```

Parameters

- **refresh** – Bool
- **groups** – str

Returns

4.4 Defaults

These are prebuilt flags and handlers for helping get information about peers and the data flow between peers.

net.**info**(**args*, ***kwargs*)

Return information about the peer requested.

```
friendly_information = net.info(peer='somepeer')
```

Returns peer.friendly_id

net.**pass_through**(**args*, ***kwargs*)

Used for testing, takes your arguments and passes them back for type testing.

```
variable = "Test this comes back the way I sent it."

response = net.pass_through(variable, peer='somepeer')
```

Returns *args, **kwargs

4.5 Peer

Each instance of python will be assigned a Peer singleton. This is not a true singleton for development and testing purposes. Although, for production, always access the peer using the `net.Peer()` call. The first thing to understand is that `net.Peer()` is referring to the Peer running in the current instance of python. So, if you are writing a connection and inside that connection you call `net.Peer()`. Depending on if that function is being run locally or remotely will determine which peer you are being returned.

net.**Peer**(**args*, ***kwargs*)

Running Peer server for this instance of python.

Returns *net.peer._Peer*

class net.peer.**_Peer**(*launch=True*, *test=False*, *group=None*)

CONNECTIONS = {b'bmV0LmRlZmF1bHRzLmhhbmRsZXJzLm51bGw=': <function null>, b'bmV0LmRlZm

SUBSCRIPTIONS = {}

FLAGS = {b'SU5WQUxJRF9DT05ORUNUSU9O': <function invalid_connection>, b'TlVMTA==': <fu

static decode(*byte_string*)
Decode a byte string sent from a peer.

Parameters **byte_string** – base64

Returns str

classmethod decode_id(*id*)
Decode a peer id

Parameters **id** – base64

Returns dict {'group': str, 'host': str, 'port': int }

classmethod encode(*obj*)
Encode an object for delivery.

Parameters **obj** – JSON compatible types

Returns str

friendly_id
Get the peers id in a friendly displayable way.

Returns str

static generate_id(*port*, *host*, *group=None*)
Generate a peers id.

Parameters

- **port** – int
- **host** – str
- **group** – str

Returns base64

classmethod get_flag(*flag*)
Get a flags id.

Parameters **flag** – str

Returns str

host
Host that the peer is running on.

Returns str

hub
Defines if this peer acts as the hub for communication through the network.

Returns bool

id
Get this peers id. This is tethered to the port and the executable path the peer was launched with. This is base64 encoded for easier delivery.

Returns base64

port
Port that the peer is running on.

Returns int

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/aldmbmtl/net/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

net could always use more documentation, whether as part of the official net docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/aldmbmtl/net/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *net* for local development.

1. Fork the *net* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/net.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv net
$ cd net/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 net tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/aldmbmtl/net/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_net
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index