net Documentation

Release 0.4.0

Alex Hatfield

Mar 25, 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.

Helpful Examples.

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

This is a very simple example of an application running on 2 different peers and communicating through a shared coding contract, the application itself.

3.2.1 app.py

```
# imports
import net


@net.connect()
def connected_function(message):
    """
    This will simply print the message passed on the local peer.
    """
    print(message)
```

3.2.2 peer1.py

```
# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app

if __name__ == '__main__':
    # enter an endless loop so the peer will listen.
    while 1:
        pass
```

3.2.3 peer2.py

```
# net imports
import net

# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app


if __name__ == '__main__':

    all_peers = net.peers(on_host=True)
    # Gets all the peers on the local network. This will return a dictionary
    # where the key is the Peer.id of the peer and value is a dictionary of
    # information about that peer. In this case, we know that both peers are
    # running on the same host so we want to pass in the no_host=True. The
    # result will be the same, but this will be significantly faster since it is
    # not going to be searching the whole network.
    #
    # Example response:
    # {
    #   'peers': {
    #       b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ==': {
    #           'group': 'None',
    #           'host': '192.168.2.24',
    #           'port': 3010,
    #           'hub': False
    #       },
```

(continues on next page)

(continued from previous page)

```
    #     },
    #    'None': [
    #        b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ=='
    #    ]
    # }

    target_peer = all_peers['None'][0]
    # since we know there is only one other peer running our application in the
    # "None" group, we can assume it is safe to grab the first key. This will be
    # how net knows where to execute our application. Just a note, by default, a
    # peer that was initialized without a group set in its environment is set to
    # the "None" group.

    app.connected_function("My Message!", peer=target_peer)
    # now we can call our applications function. Since this function is
    # connected through net, we can pass in the keyword argument 'peer' to
    # specify where to execute the function. If you do not specify the peer, it
    # will simply execute the code locally as though it was not connected.
    # Basically just a normal function.

    # Running this will now print "My Message!" on peer1.
```

CHAPTER 4

Examples

4.1 Connection

The files required for this example are:

- app.py
- peer1.py
- peer2.py

4.1.1 app.py

```
# imports
import net


@net.connect()
def connected_function(message):
    """
    This will simply print the message passed on the local peer.
    """
    print(message)
```

4.1.2 peer1.py

```
# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    # enter an endless loop so the peer will listen.
    while 1:
        pass
```

4.1.3 peer2.py

```
# net imports
import net

# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app


if __name__ == '__main__':

    all_peers = net.peers(on_host=True)
    # Gets all the peers on the local network. This will return a dictionary
    # where the key is the Peer.id of the peer and value is a dictionary of
    # information about that peer. In this case, we know that both peers are
    # running on the same host so we want to pass in the no_host=True. The
    # result will be the same, but this will be significantly faster since it is
    # not going to be searching the whole network.
    #
    # Example response:
    # {
    #   'peers': {
    #       b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ==': {
    #           'group': 'None',
    #           'host': '192.168.2.24',
    #           'port': 3010,
    #           'hub': False
    #       },
    #    },
    #   'None': [
    #       b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ=='
    #   ]
    # }

    target_peer = all_peers['None'][0]
    # since we know there is only one other peer running our application in the
    # "None" group, we can assume it is safe to grab the first key. This will be
    # how net knows where to execute our application. Just a note, by default, a
    # peer that was initialized without a group set in its environment is set to
    # the "None" group.

    app.connected_function("My Message!", peer=target_peer)
    # now we can call our applications function. Since this function is
    # connected through net, we can pass in the keyword argument 'peer' to
    # specify where to execute the function. If you do not specify the peer, it
    # will simply execute the code locally as though it was not connected.
    # Basically just a normal function.

    # Running this will now print "My Message!" on peer1.
```

4.2 Tagged Connection

The files required for this example are:

- app_v1.py
- app_v2.py
- peer1.py
- peer2.py
- peer3.py

4.2.1 app_v1.py

```
# imports
import os

# configure our net configuration with a group identifier
os.environ['NET_GROUP'] = 'app_v1'

# net imports
import net


# application code version 1
@net.connect("myTaggedFunction")  # <- this can be any value.
def connected_function(message):
    """
    This will return the message with the " Version 1" appended to the end.
    """
    return message + ' Version 1'
```

4.2.2 app_v2.py

```
# imports
import os

# configure our net configuration with a group identifier
os.environ['NET_GROUP'] = 'app_v2'

# net imports
import net


# application code version 2
@net.connect("myTaggedFunction")
def connected_function(message):
    """
    This will return the message with the " Version 2" appended to the end.
    """
    return message + ' Version 2'
```

4.2.3 peer1.py

```
# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app_v1

if __name__ == '__main__':
    # enter an endless loop so the peer will listen.
    while 1:
        pass
```

4.2.4 peer2.py

```
# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app_v2

if __name__ == '__main__':
    # enter an endless loop so the peer will listen.
    while 1:
        pass
```

4.2.5 peer3.py

```
# net imports
import net

# This version of the peer will run the latest version of the
import app_v2


if __name__ == '__main__':

    all_app_peers = net.peers(groups=['app_v1', 'app_v2'], on_host=True)
    # First we need to grab the to running peers on our local host in both the
    # app_v1 and app_v2 groups. This will give us a dict laid out as follows.
    # {
    #       'peers': {
    #           peer1_id: info,
    #           peer2_id: info,
    #       },
    #       app_v1: [
    #           peer1_id
    #       ],
    #       app_v2: [
    #           peer2_id
    #       ]
    # }
    # This will allow us to better access the peers. You can grab a peer
    # directly of the dictionary OR get the group which has a list of all the
    # peer_ids that belong to it. Then use that list of peers to gran the
    # information.
```

(continues on next page)

(continued from previous page)

```
    app_v1_peer = all_app_peers['app_v1'][0]
    # Lets grab the first peer that is using the app_v1 api and execute our
    # tagged function

    response = app_v2.connected_function("My Message!", peer=app_v1_peer)
    print(response)
    # This will result with "My Message! Version 1". Which shows that this new
    # api_v2 can still request a tagged version on an older platform.

    app_v2_peer = all_app_peers['app_v2'][0]
    # Lets grab the first peer that is using the app_v2 api and execute our
    # tagged function

    response = app_v2.connected_function("My Message!", peer=app_v2_peer)
    print(response)
    # This will result with "My Message! Version 2". Which is the latest version
    # of the api.
```

4.3 Subscription

The files required for this example are:

- app.py
- hub.py
- peer1.py
- peer2.py

You will need to launch the `hub.py` before you launch the peers. After you launch the hub and however many peers you want, enter in your message and it will be echoed on all of the peers. A big thing to note is that the peers will not connect if the hub isn't up. This is because subscriptions only happen when the peer is launched and there are no re-tries. So any peer launched before the hub will not get the event triggers.

Feel free to shut down the peers and try to enter your message again. You will see the hub will not error. It will simply ignore the missing peers. This will happen if the peers were to fail and error as well. The hub will ignore and just continue on with its own execution.

4.3.1 app.py

```
# configure this application as having its own network group through the
# environment set up
import os

# our peers will now all belong to the group, 'myApp'
os.environ['NET_GROUP'] = 'myApp'

# net imports
import net


@net.event("myEvent")  # <- this can be any value.
def something_happened(*args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
    return args, kwargs


# A subscription allows you to connect to an event on another peer. This does
# not need to always be a hub and peers can subscribe to events on any other
# peer the same way we did here minus the "hubs_only=True"
@net.subscribe("myEvent", hubs_only=True, on_host=True)
def handle_something_happened(message):
    """
    Simply print what happened.
    """
    print(message)
```

4.3.2 hub.py

```
# configure this as the hub of the group
import os

os.environ['NET_IS_HUB'] = 'True'

# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app

if __name__ == '__main__':
    print("Enter the message to send to the peers subscribed to you.")
    while 1:
        # As you can imagine, this can be used anywhere in your application. In
        # this example, we are just going to take your message and broadcast it
        # to all the subscribed peers.
        your_message = input("Message: ")

        # This will trigger the "myEvent" that was wrapped on around this
        # function. When this is triggered, it will package up your message and
        # send it to the peers.
        app.something_happened(your_message)
```

4.3.3 peer1.py

```
# Importing the application code will automatically launch the peer and begin
# listening for connection requests as well as set up the connection registry.
import app

if __name__ == '__main__':
    # enter an endless loop so the peer will listen.
    while 1:
        pass
```

CHAPTER 5

API Reference

5.1 Environment

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

5.1.1 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.

5.1.2 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.

5.1.3 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.

5.1.4 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.

5.2 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*, *groups=None*, *hubs_only=False*, *peers=None*, *on_host=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.**event**(*name*)

Registers a function as an event trigger. Event triggers are hooks into the event system between peers. Peers that `net.subscribe` to a peer, register an event on that peer.

Lets say PeerA subscribes to an event on PeerB using the following code.

```
# code on PeerA

peerB_id = "peerb"

@net.subscribe("doing_something")
def handleEvent(whatPeerBDid):
    ...do something
```

The subscribe decorator has communicated with PeerB and registered itself as on the list of peer to update if "doing_something" is ever triggered. On PeerB's side we have the following.

```
# code on PeerB

@net.event("doing_something")
def imDoingSomething(*args, **kwargs):
    return args, kwargs
```

Note: All functions flagged as an event MUST return args and kwargs exactly as displayed above.

Now lets say in PeerB we want to trigger the event in a for loop and have it hand off the values to all the subscribed peers, PeerA in this case.

```
for i in range(0, 10):
    imDoingSomething(i)  # <- this will notify PeerA and pass the value of 'i'.
```

Keep in mind, you can have any number of peers subscribe to any kind of event. So if we had 5 peers subscribe to PeerB they would all be passed this value at runtime.

Lastly, these event functions act as a buffer between the runtime code of your application and the delivery of the content to the peer. For example:

```
var = MyCustomObject()  # some JSON incompatible object

...do a bunch of delivery prep and muddy up the application code...

imDoingSomething(var)
```

Instead

```
@net.event("doing_something")
def imDoingSomething(*args, **kwargs):

    obj = args[0]

    ...clean and prepare for transit here...

    args[0] = cleanedObj

    return args, kwargs
```

As you can see, these functions act as a hook into the delivery system when the event is triggered.

There are protections put in place to try to prevent the peer that triggered the event to be blocked by a bad handle on the subscribed peer. For the purpose of protecting the event triggering peer from remote errors, all connection errors and remote runtime errors will be caught and logged. But nothing will halt the running application.

i.e. event -> remote peer errors -> event peer will log and ignore

Stale peer subscriptions will be added to the stale list and pruned. Since the subscriptions are created per client request, the event peer will not know until a request is made that the subscribed peer went offline.

net.**flag**(*name*)
Register a function as a flag handler for the peer server.

Parameters **name** – str

5.3 Functions

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

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

Get a list of all peers on your network. This is a cached values since the call to graph the network can be long. You can also limit this search to only look for operating peers on the localhost which does not require the long network scan, just set the `on_host` kwarg to True.

Hubs act as the centers for certain application events or processes. In some cases, you may only want to subscribe or communicate with hubs. You can specify this through the `hubs_only` kwarg.

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()
```

Only search for peers on local host and not on the network.

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

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)
```

Refresh the cache with peers in group1 and 2

```
all_peers = net.peers(["group1", "group2"], refresh=True)
```

Refresh the cache with all of the hubs on the network regardless of group.

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

Refresh the cache with only hubs in group1 and 2

```
all_peers = net.peers(["group1", "group2"], hubs_only=True, refresh=True)
```

Parameters

- **refresh** – Bool
- **groups** – str
- **on_host** – Bool
- **hubs_only** – Bool

Returns

{

Peers 'peers': {

b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ==': { 'group': 'None', 'host': '192.168.2.24', 'port': 3010, 'hub': False, 'executable': path/to/executable, 'user': username

},

},

Groups 'None': [

b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ=='

]

}

5.4 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

5.5 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

static decode_id(*id*)
Decode a peer id

Parameters **id** – base64

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

static 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

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

5.6 Full Package

5.6.1 net

net package

Subpackages

net.defaults package

Submodules

net.defaults.flags module

Default Flags

Prebuilt flags for net. Do not modify.

net.defaults.flags.**null_response**(*connection*, *foreign_peer_id*)
Execute this if the peer has returned the NULL_RESPONSE flag.

Parameters

- **connection** – name of the connection requested
- **foreign_peer_id** – The foreign peers friendly_id

Returns str

net.defaults.flags.**invalid_connection**(*connection*, *foreign_peer_id*)
Execute this if the peer has returned the NULL_RESPONSE flag.

Parameters

- **connection** – name of the connection requested
- **foreign_peer_id** – The foreign peers friendly_id

Returns

net.defaults.handlers module

Default Connected Handlers

Prebuilt connected handlers for net. Do not modify.

net.defaults.handlers.**info**(**args*, ***kwargs*)
Return information about the peer requested.

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

Returns peer.friendly_id

net.defaults.handlers.**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

net.defaults.handlers.**null**(**args*, ***kwargs*)
Return a null response flag

Returns NULL Flag

net.defaults.handlers.**subscription_handler**(*event*, *peer*, *connection*)
Will register the incoming peer and connection with the local peers subscription of the event passed. This is for internal use only.

Parameters

- **event** – event id
- **peer** – foreign peer id
- **connection** – connection id

net.defaults.handlers.**connections**(**args*, ***kwargs*)
Return the connections registered with the peer.

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

Returns peer.CONNECTIONS

Module contents

Submodules

net.api module

api module

Contains the general network interactions for net.

net.api.**peers**(*refresh=False*, *groups=None*, *on_host=False*, *hubs_only=False*)
Get a list of all peers on your network. This is a cached values since the call to graph the network can be long. You can also limit this search to only look for operating peers on the localhost which does not require the long network scan, just set the `on_host` kwarg to True.

Hubs act as the centers for certain application events or processes. In some cases, you may only want to subscribe or communicate with hubs. You can specify this through the `hubs_only` kwarg.

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()
```

Only search for peers on local host and not on the network.

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

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)
```

Refresh the cache with peers in group1 and 2

```
all_peers = net.peers(["group1", "group2"], refresh=True)
```

Refresh the cache with all of the hubs on the network regardless of group.

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

Refresh the cache with only hubs in group1 and 2

```
all_peers = net.peers(["group1", "group2"], hubs_only=True, refresh=True)
```

Parameters

- **refresh** – Bool
- **groups** – str
- **on_host** – Bool
- **hubs_only** – Bool

Returns

{

Peers 'peers': {

b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ==': { 'group': 'None', 'host': '192.168.2.24', 'port': 3010, 'hub': False, 'executable': path/to/executable, 'user': username

},

},

Groups 'None': [

b'MTkyLjE2OC4yLjI0OjMwMTAgLT4gTm9uZQ=='

]

}

net.cli module

Console script for net.

net.connect module

Connect Module

Contains the connect decorator and should have nothing else.

net.connect.**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.environment module

Handler Module

Contains the peer handler and should have nothing else.

net.event module

Event Module

Contains the event decorator and should have nothing else.

net.event.**event**(*name*)

Registers a function as an event trigger. Event triggers are hooks into the event system between peers. Peers that `net.subscribe` to a peer, register an event on that peer.

Lets say PeerA subscribes to an event on PeerB using the following code.

```
# code on PeerA

peerB_id = "peerb"

@net.subscribe("doing_something")
def handleEvent(whatPeerBDid):
    ...do something
```

The subscribe decorator has communicated with PeerB and registered itself as on the list of peer to update if "doing_something" is ever triggered. On PeerB's side we have the following.

```
# code on PeerB

@net.event("doing_something")
def imDoingSomething(*args, **kwargs):
    return args, kwargs
```

Note: All functions flagged as an event MUST return args and kwargs exactly as displayed above.

Now lets say in PeerB we want to trigger the event in a for loop and have it hand off the values to all the subscribed peers, PeerA in this case.

```
for i in range(0, 10):
    imDoingSomething(i)  # <- this will notify PeerA and pass the value of 'i'.
```

Keep in mind, you can have any number of peers subscribe to any kind of event. So if we had 5 peers subscribe to PeerB they would all be passed this value at runtime.

Lastly, these event functions act as a buffer between the runtime code of your application and the delivery of the content to the peer. For example:

```
var = MyCustomObject()  # some JSON incompatible object

...do a bunch of delivery prep and muddy up the application code...

imDoingSomething(var)
```

Instead

```
@net.event("doing_something")
def imDoingSomething(*args, **kwargs):

    obj = args[0]

    ...clean and prepare for transit here...

    args[0] = cleanedObj

    return args, kwargs
```

As you can see, these functions act as a hook into the delivery system when the event is triggered.

There are protections put in place to try to prevent the peer that triggered the event to be blocked by a bad handle on the subscribed peer. For the purpose of protecting the event triggering peer from remote errors, all connection errors and remote runtime errors will be caught and logged. But nothing will halt the running application.

i.e. event -> remote peer errors -> event peer will log and ignore

Stale peer subscriptions will be added to the stale list and pruned. Since the subscriptions are created per client request, the event peer will not know until a request is made that the subscribed peer went offline.

net.flag module

Flag Module

Contains the flag decorator and should have nothing else.

net.flag.**flag**(*name*)
Register a function as a flag handler for the peer server.

Parameters **name** – str

net.handler module

Handler Module

Contains the peer handler and should have nothing else.

class net.handler.**PeerHandler**(*request*, *client_address*, *server*)
Bases: socketserver.BaseRequestHandler

Handles all incoming requests to the applications Peer server. Do not modify or interact with directly.

handle()
Handles all incoming requests to the server.

net.imports module

python 2/3 imports handled here

exception net.imports.**ConnectionRefusedError**
Bases: ConnectionError

Connection refused.

net.peer module

net.peer.**Peer**(**args*, ***kwargs*)
Running Peer server for this instance of python.

Returns *net.peer._Peer*

net.subscribe module

Subscribe Module

Contains the subscribe decorator and should have nothing else.

net.subscribe.**subscribe**(*event*, *groups=None*, *hubs_only=False*, *peers=None*, *on_host=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
```

Module contents

Top-level package for net.

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.**flag**(*name*)
Register a function as a flag handler for the peer server.

Parameters **name** – str

net.**Peer**(**args*, ***kwargs*)
Running Peer server for this instance of python.

Returns *net.peer._Peer*

net.**null_response**(*connection*, *foreign_peer_id*)
Execute this if the peer has returned the NULL_RESPONSE flag.

Parameters

- **connection** – name of the connection requested
- **foreign_peer_id** – The foreign peers friendly_id

Returns str

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

net.**null**(**args*, ***kwargs*)
Return a null response flag

Returns NULL Flag

net.**info**(**args*, ***kwargs*)
Return information about the peer requested.

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

Returns peer.friendly_id

net.**invalid_connection**(*connection*, *foreign_peer_id*)
Execute this if the peer has returned the NULL_RESPONSE flag.

Parameters

- **connection** – name of the connection requested
- **foreign_peer_id** – The foreign peers friendly_id

Returns

net.**subscribe**(*event*, *groups=None*, *hubs_only=False*, *peers=None*, *on_host=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.**event**(*name*)

Registers a function as an event trigger. Event triggers are hooks into the event system between peers. Peers that net.subscribe to a peer, register an event on that peer.

Lets say PeerA subscribes to an event on PeerB using the following code.

```
# code on PeerA

peerB_id = "peerb"

@net.subscribe("doing_something")
def handleEvent(whatPeerBDid):
    ...do something
```

The subscribe decorator has communicated with PeerB and registered itself as on the list of peer to update if "doing_something" is ever triggered. On PeerB's side we have the following.

```
# code on PeerB

@net.event("doing_something")
def imDoingSomething(*args, **kwargs):
    return args, kwargs
```

Note: All functions flagged as an event MUST return args and kwargs exactly as displayed above.

Now lets say in PeerB we want to trigger the event in a for loop and have it hand off the values to all the subscribed peers, PeerA in this case.

```
for i in range(0, 10):
    imDoingSomething(i)  # <- this will notify PeerA and pass the value of 'i'.
```

Keep in mind, you can have any number of peers subscribe to any kind of event. So if we had 5 peers subscribe to PeerB they would all be passed this value at runtime.

Lastly, these event functions act as a buffer between the runtime code of your application and the delivery of the content to the peer. For example:

```
var = MyCustomObject()  # some JSON incompatible object

...do a bunch of delivery prep and muddy up the application code...

imDoingSomething(var)
```

Instead

```
@net.event("doing_something")
def imDoingSomething(*args, **kwargs):

    obj = args[0]

    ...clean and prepare for transit here...

    args[0] = cleanedObj

    return args, kwargs
```

As you can see, these functions act as a hook into the delivery system when the event is triggered.

There are protections put in place to try to prevent the peer that triggered the event to be blocked by a bad handle on the subscribed peer. For the purpose of protecting the event triggering peer from remote errors, all connection errors and remote runtime errors will be caught and logged. But nothing will halt the running application.

i.e. event -> remote peer errors -> event peer will log and ignore

Stale peer subscriptions will be added to the stale list and pruned. Since the subscriptions are created per client request, the event peer will not know until a request is made that the subscribed peer went offline.

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

Return the connections registered with the peer.

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

Returns peer.CONNECTIONS

CHAPTER 6

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:

6.1 Types of Contributions

6.1.1 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.

6.1.2 Fix Bugs

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

6.1.3 Implement Features

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

6.1.4 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.

6.1.5 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 :)

6.2 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.

6.3 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.

6.4 Tips

To run a subset of tests:

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

6.5 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 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

C

D

E

F

G

H

I

N

P

S