peerpy

Jul 02, 2020

Table of contents

This module provides a **high-level API** for discovering and connecting multiple devices on a local network, without the headache of implementing python's built-in socket module!

Note: Peerpy is built on top of the python's builtin `socket` module, using `threading` for parallel computing, naming:

- your application code
- listening for connections
- sending/receiving data
- sending/answering pings

Note: This module allows to quickly link the application layer (your application) to the transport layer (provided by python's `socket` module) of the OSI model.

CHAPTER 1

What is it made for?

- IoT devices (e.g. Rasberry Pi)
- Blockchain
- Fast proof of concepts

CHAPTER 2

Installation

```
pip install peerpy
```

CHAPTER 3

Source code

Please refer to python sources documentation or peerpy's github repository.

3.1 User manual

3.1.1 General introduction

A `Peer` is an instance of a listening device, which is able to connect to another listening `Peer`. A `Connection` represents a link between 2 `Peer` on a local network. It allows the serialization and the exchange of data over TCP.

Note: Every address is an IPv4 address.

Note: Every connection has 2 fixed peers, 1 fixed data type and possibly 1 fixed data size in case of a streaming connection (a connection over which we only exchange data of fixed size).

Note: Data type is one of:

- `"raw"` (any object pickle-serialized to bytes)
- `"json"` (any json-serializable object)
- `"bytes"` (explicit)

3.1.2 Protocols

Connection protocol

In the following scenario, Alice knows the address and listening port of Bob:

- Alice sends a *HELLO* header to Bob, containing her address and port and information about the desired connection: **HELLO|address_name=127.0.0.1:51515&data_type=json**
- Bob receives the *HELLO* header and answers with a *ACCEPT* or *DENY* header, depending on his choice.
- Alice waits for an answer from Bob within a given timeout, before giving up. If she receives a *ACCEPT* header from Bob, both acknowledge they have established a connection.

Note: For the underlying python's `socket`, a connection is established from the first step of this protocol. As we only listen for utf-8 encoded bytes headers until the end of the previous protocol, this doesn't present security issues.

Data protocol

In the following scenario, Alice and Bob already established a connection and thus have come to an agreement on data types for this connection (and possible data size, in case of a streaming connection). Alice wants to send Bob some data:

- Alice sends a *DATA* header to Bob, containing information about the following data: **DATA|data_size=2048&data_type=raw**
- Bob receives the header and reads it: he sees that the following data_type is `raw`. If they previously agreed on a `strict` connection, Bob shuts down the connection with Alice, as Alice violated their agreement. Otherwise, he proceeds with receiving data.
- As TCP is a reliable data exchange protocol, no further acknowledgment packet is exchanged and the data transmission is considered completed.

Discovery protocol

Peerpy comes with a builtin discovery protocol built over UDP. In the following scenario, Alice wants to discover people on her local network:

- Alice sends a *PING* packet to her router's UDP broadcasting IPv4 address, containing her address and listening port: **PING 192.168.0.2:51515**.
- Bob listens for packets on his router's UDP broadcasting IPv4 address, waiting for *PING* packets. He receives Alice's packet and sends her a *PONG* packet, containing his address and listening port: **PING 192.168.0.3:62626**.
- Alice receives Bob's *PONG* packet and thus knows that Bob is reachable over the address he shared.

3.1.3 Events & Handlers

`Peer` and `Connection` classes both inherits from the `EventHandler` superclass, which allows one to pass event handlers (python callables) which will respectively be called by the peer's listening thread and the connection's main thread upon the corresponding event.

Let's say for example that you want to print your `Peer` object's address and listening port (which is the default behavior). Then you just have to register your handler at your `Peer` instanciation:

```
with Peer(handlers={
   "listen": lambda peer: print(peer.address, peer.address_name)
}) as peer:
```

Here is a table showing every events and handlers possible:

Changed in version 1.2.0: Whenever a handler is called, the first argument is now the event emitter.

Emitter	Event	Description	Additional arguments
`Peer`	`listen`	Triggered when peer is listening for connections	
	`offer`	Triggered when peer has received a connection offer. This handler must return a boolean indicating whether to accept or deny the offer.	The connection to accept or deny
	`connection`	Triggered when peer has established a new connection	The connection established
	`stop`	Triggered when peer has stopped listening for connections	
`Connection`	`data`	Triggered when connection has received some data	The data received
	`close`	Triggered when connection has been terminated	

3.2 Python sources

3.2.1 Connection

class Connection(*peer*, *target_name: str*, *sock: socket.socket*, *buffer_size: int*, ***kwargs*)

Bases: *peerpy.event_handler.EventHandler*

close(*force: bool = False*)

Closes the connection nicely.

Parameters

- **force** (*bool, optional*) – whether to force close the connection.
- **setting should be considered dangerous, as data can be lost. Defaults to False.** (*This*) –

closed

Returns whether this connection is closed.

Returns a boolean indicating whether the connection is closed.

Return type bool

data_type

Returns the data type each peers have agreed on for this connection.

Returns the string representation of the data type

Return type str

send(*data: Any*)

Send data to the target peer, serializing it to this connection's default data format.

Parameters **data** (*Any*) – the data to serialize and send.

Raises `ValueError` – if this connection's default format is bytes and the data is not bytes

Returns whether data was successfully sent.

Return type bool

start_thread()
Attempts to start this connection's main thread, if not already running.

3.2.2 Data

class Data(*_type: str*, *buffer: bytes = b''*, *decoded_data: Any = None*, *encoded_data: bytes = None*)
Bases: object

buffer = b''

decode()

decoded_data = None

encode()

encoded_data = None

get_type()

3.2.3 EventHandler

class EventHandler(*event_names: List[str]*, *handlers: Dict[str, Callable[[Any], Any]]*, *min_handler_names: List[str] = None*)
Bases: object

Super class that registers and handle events, for objects such as Peer and Connection.

handle(*event_name: str*, **args*) → Any
Calls a handler for a specific event if existing, passing it arguments.

Parameters **event_name** (*str*) – the event to trigger.

Returns whatever the handler, if existing, returns

Return type Any

set_handler(*handler_type: str*, *handler: Callable*)
Sets a callable as an event handler.

Parameters

- **handler_type** (*str*) – the event name.
- **handler** (*Callable*) – the handler to be called when event is triggered.

wait(*event_name: str*, *timeout: float = None*) → Any
Waits for an event to trigger and returns the handler's return value.

Parameters

- **event_name** (*str*) – the event to wait for.
- **timeout** (*float, optional*) – how long maximum to wait for the event, in seconds. Defaults to None.

Raises

- ValueError – if event_name is not a valid event name for this handler.
- HandlerMissingException – if no handler is registered for the event, while it is a necessary handler.

Returns whatever the handler returns.

Return type Any

3.2.4 Exceptions

exception DataSizeError
Bases: Exception

Raised the data size doesn't correspond to the connection's data size.

exception DataTypeError
Bases: Exception

Raised when data type doesn't correspond to the connection's data type.

exception HandlerMissingException
Bases: Exception

Raised when an event handler is missing a handler for a specific event.

exception HeaderSizeError
Bases: Exception

Raised when the header's size is greater than the protocol's header size.

3.2.5 Peer

class Peer(*address: str = None*, *port: int = 0*, ***kwargs*)
Bases: *peerpy.event_handler.EventHandler*

address
This peer' address, in a normalized format

Returns the normalized address (ipv4, port)

Return type Tuple[str, int]

address_name
This peer's normalized address name

Returns the normalized address name ipv4:port

Return type str

broadcast(*data: Any*)
Broadcasts data to all the connected peers.

Parameters **data** (*Any*) – the data to broadcast

connect(*address: str*, *port: int = None*, *data_type: str = 'json'*, *strict: bool = True*, ***kwargs*) → peerpy.connection.Connection
Attempts to start a connection with a remote peer located at (address, port). Additional arguments are passed to the Connection constructor and sent to the remote peer right after successful connection, so that it knows with what data type to communicate with.

Parameters

- **address** (*str*) – the ipv4 address of the remote peer, provided with the port if wanted (ipv4:port)

- **port** (*int, optional*) – the port to use for the connection, if not provided in address. Defaults to None.
- **data_type** (*str, optional*) – the data type to use for the connection. Defaults to "raw".
- **strict** (*bool, optional*) – whether this connection is strict on data types. Defaults to True.
- **buffer_size** (*int, optional*) – the buffer size to use to receive data. Defaults to this peer's buffer size.

Returns the connection, if established

Return type *Connection*

get_local_peers() → List[str]
Returns the list of peers visible on the same local network.

Returns the list of visible peers' addresses

Return type List[str]

invisible
Whether this peer is invisible to other peers on the same local network

Returns this peer's invisibility

Return type bool

start()
Attempts to start this peer's server and pinger (if needed).

stop(*_async=False*)
Attempts to stop this peer and all its connections.

Parameters **_async** (*bool, optional*) – whether to stop this peer asynchronously. Defaults to False.

timeout
This peer's default timeout

Returns the default timeout

Return type float

3.2.6 Protocol

class Defaults(*buffer_size: int = 8192*, *timeout: float = 2*, *peer_handlers: Dict[str, Callable] = <factory>*, *connection_handlers: Dict[str, Callable] = <factory>*)
Bases: object

buffer_size = 8192

timeout = 2

class Headers(*size: int = 128*, *separator: str = '|'*, *values_separator: str = '&'*, *key_separator: str = '='*, *data_header: str = 'DATA'*, *hello_header: str = 'HELLO'*, *accept_header: str = 'ACCEPT'*, *deny_header: str = 'DENY'*, *ping_header: str = 'PING'*, *pong_header: str = 'PONG'*, *data_types_parsers: Dict[str, Callable] = <factory>*, *required_hello_fields: List[str] = <factory>*, *required_data_fields: List[str] = <factory>*)
Bases: object

accept_header = 'ACCEPT'

```
data_header = 'DATA'
deny_header = 'DENY'
hello_header = 'HELLO'
key_separator = '='
ping_header = 'PING'
pong_header = 'PONG'
separator = '|'
size = 128
values_separator = '&'
```

Python Module Index

p

Index