packetmq Documentation

Release 1.0

notna

July 17, 2016

Contents

Contents:

CHAPTER 1

Getting Started

This is a simple guide to the most important features of packetmq.

1.1 Installation

Installation with pip:

```
$ pip install packetmq
```

Installation with easy_install:

```
$ easy_install packetmq
```

You can also download and manually install packetmq here.

If installing manually, do not forget to also install twisted, u-msgpack-python and bidict.

1.2 The Packet Registry

The `PacketRegistry` is used to store packet objects, name and numid.

First, we need to import packetmq:

```
>>> import packetmq
```

Then, we can create the `PacketRegistry` instance:

```
>>> reg = packetmq.PacketRegistry()
>>> reg.registerDefaultPackets() # Initializes all standard packets required for normal operation
```

We can then access the default packets using the so called *Auto-Types* that convert between different representations of a packet smartly:

```
>>> reg.packetInt("packetmq:handshake_init") # packetInt converts all representations to a numid
0
>>> reg.packetStr(0) # packetStr converts all representations to the packets name
"packetmq:handshake_init"
>>> reg.packetObj("packetmq:handshake_init") # packetObj converts all representations to the packet o
<packetmq.packet.HandshakeInitPacket object at 0x.........>
>>> reg.packetInt(0) # if the type is already correct, conversion is skipped
0
```

To add new packets, we just call `packetmq.PacketRegistry.addPacket()` with the name, object and numid:

```
>>> mypacket = packetmq.packet.PrintPacket()
>>> reg.addPacket("myapplication:mypacket",mypacket,17) # the number needs to be above 16 and below
>>> reg.packetObj("myapplication:mypacket")
<packetme.packet.EchoPacket object at 0x........>
>>> reg.packetInt(mypacket)
17
```

You can also create new packets by subclassing `packetmq.packet.Packet`.

1.3 Sending Data to Peers

Now, that we know how to use the PacketRegistry, we can move on to sending actual data to the server or client. For now, we will setup a simple echo system:

```
>>> mypacket = packetmq.packet.EchoPacket(retType="myapplication:myprintpacket") # EchoPacket simply
>>> myprintpacket = packetmq.packet.PrintPacket() # PrintPacket prints out the packet
>>> reg.addPacket("myapplication:mypacket",mypacket,17)
>>> reg.addPacket("myapplication:myprintpacket",myprintpacket,18)
```

Then, we create the server in one session:

```
[server]>>> server = packetmq.Server(reg)
[server]>>> server.listen(12345)
[server]>>> server.runAsync()
```

For TCP clients, the client should not run in the same process, just use a new shell and do all the above steps for packet registration, but set the argument adaptPacketIds to True in the PacketRegistry:

```
[client]>>> client = packetmq.Client(reg)
[client]>>> client.connect(("localhost",12345)) # change the address to the server's IP address, if
[client]>>> client.runAsync()
```

Now, both peers are connected and you can start transmitting data using `sendPacket()`

```
[client]>>> client.sendPacket({"foo":"bar",123:None,0.001:True,"mylist":["abc","def"]},"myapplicatio
[client]>>> {"foo":"bar",123:None,0.001:True,"mylist":["abc","def"]} # Note that this output will be
[server]Received EchoPacket: {"foo":"bar",123:None,0.001:True,"mylist":["abc","def"]} # Printed on th
```

You could also send packets from the server to the client or maybe you want to communicate between threads, then you can use `packetmq.MemoryServer` and `packetmq.MemoryClient`

1.4 Creating new packet types

Coming soon, for now look at the sources on github if you want information about creating new packet types.

CHAPTER 2

packetmq - Packet based networking

2.1 Packet Registry

PacketRegistry([adaptPacketIds)]:
Packet Registry used by both server and client.

adaptPacketIds defines whether to adapt or enforce packet IDs when connecting, usually you set this to *False* on servers or to *True* on clients.

addPacket(name,obj,numid[,bypass_assert)]:
Registers packet Object *obj* using name *name* and numerical id *numid*.

name should be of format *"<application>:<packet>"* and should only contain standard ascii chars. *name* must also be unique.

obj is an instance if `packetmq.packet.Packet` or subclass. You may in theory use the same instance for multiple packets, even though not very usefull.

numid is an int within `MIN_PACKET_ID` and `MAX_PACKET_ID`, inclusive. This int represents the packet type on the wire.

bypass_assert may be used by internal packets that need to bypass the numerical id limitations.

delPacket(arg):
Removes packet *arg* from the registry.

arg can be any packet type, see *Auto-Types*.

packetStr(arg):
packetInt(arg):
packetObj(arg):
Auto-Types conversion methods for packet objects.

These methods allow conversion between `packetmq.packet.Packet`, Numerical IDs and names.

registerDefaultPackets():
Registers all default packets needed by the handshake and other default functionality.

2.2 Peer Base Class

Peer(registry[,proto[,factory)]]:
Base Class for peers in communication.

registry must be an instance of `PacketRegistry` or subclass.

proto and *factory* are used for creating new connections. You normally do not need to change these.

peerFileno(arg):
peerObj(arg):
Auto-Types conversion methods for protocol/connection objects.

These methods allow conversion between either `packetmq.packetprotocol.PacketProtocol` for TCP connections or `packetmq.Peer` for memory connections and Connection IDs.

initConnection(conn):
Initializes connection *conn*, e.g. sends the handshake.

This method is called automatically by `PacketProtocol.connectionMade()` and thus should not be called.

lostConnection(conn[reason)]:
Callback called when the connection with *conn* is lost.

reason is either a dotted string describing the reason or a reason given by twisted. If a dotted string is passed, usually a softquit has occured and when a reason by twisted is passed, then the connection was aborted.

This callback is the last chance to send another packet to the peer, however responses may not arrive.

sendPacket(data,dtype,to):
Sends a packet of type *dtype* to peer *to* with payload *data*.

dtype can be any packet type, see *Auto-Types*.

to can be any peer type, see *Auto-Types*.

data can be of any type, by default only msgpack-compatible objects are accepted. Accepted values can be changed by the packet.

This methods encodes and frames the data and sends it with `sendEncoded()`.

sendEncoded(raw,to):
Sends the raw data *raw* to peer *to*.

raw can be any string, including special characters.

to can be any peer type, see *Auto-Types*.

Data is sent either through TCP or memory.

recvPacket(data,dtype,fromid):
Called to process packets.

data is the decoded data, e.g. most often dicts or lists.

dtype can be any packet type, see *Auto-Types*.

fromid can be any peer type, see *Auto-Types*.

This method will be called automatically by `recvEncoded()`.

recvEncoded(data,fromid):
Called by twisted's reactor methods upon receiving full packets.

data is the encoded data, e.g. most often msgpack encoded data.

fromid can be any peer type, see *Auto-Types*.

This method is called automatically and thus should not be called manually.

run():
Starts the reactor in the same thread. The reactor processes all incoming and outcoming network traffic.

This call blocks until `Peer.stop()` is called.

runAsync():
Calls `Peer.run()` in another thread.

This call does not block, but you will still need to call `Peer.stop()`, else your program will continue running infinitely.

If the main loop is started using this method, spawning subprocesses via twisted will not work, because their termination cannot be detected.

stop():
Stops the reactor and all traffic processing without closing the connections.

This is also called when the peer gets deleted.

softquit(peer[,reason)]:
Soft-closes the connection to peer *peer*, optionally with the reason *reason*.

This will also trigger `Peer.lostConnection()`.

on_connMade(conn):
Callback called when connection *conn* is made.

conn can be any peer type, see *Auto-Types*.

2.3 TCP Servers and Clients

Server(registry[,proto[,factory]]):
TCP Server class powered by twisted. This class is a subclass of `Peer`.

registry must be an instance of `PacketRegistry` or subclass.

proto and *factory* are used for creating new connections. You normally do not need to change these.

listen(port):
Listen to TCP port *port*.

You can call this method multiple times to listen to multiple ports.

Client(registry[,proto[,factory)]:
TCP Client class powered by twisted. This class is a subclass of `Peer`.

registry must be an instance of `PacketRegistry` or subclass.

proto and *factory* are used for creating new connections. You normally do not need to change these.

Most methods that require a peer will default to the first connected server. This applies to following methods:

- `sendPacket()`
- `sendEncoded()`
- `recvPacket()`
- `recvEncoded()`

connect(address):
Connects to TCP address tuple *address*.

address is a tuple of *(host,port)*.

2.4 Memory Servers and Clients

All memory servers and clients also have `setState()` and `getState()` state methods, for compatibility with `PacketProtocol`.

`MemoryServer(registry[,proto[,factory]])`:
Memory Server class for in-process data transmission. This class is a subclass of `Peer`.

registry must be an instance of `PacketRegistry` or subclass.

proto is not used since all connections are in-memory.

factory is used for storing active connections.

`connectClient(client)`:
Connects client *client* to the server.

This method should not be used manually, use `MemoryClient.connect()` instead.

`disconnectClient(client)`:
Disconnects client *client* from the server.

This method should not be used manually, use `MemoryClient.disconnect()` instead.

`MemoryClient(registry[,proto[,factory]])`:
Memory Client class for in-process data transmission. This class is a subclass of `Peer`.

registry must be an instance of `PacketRegistry` or subclass.

proto is not used since all connections are in-memory.

factory is used for storing active connections.

`connect(server)`:
Connects the client with the server *server*.

`disconnect([server])`:
Terminates the connection to server *server* and calls all appropriate callbacks.

CHAPTER 3

Auto-Types

Auto-types are a mechanic supported by several classes in packetmq.

Auto-Types allows you to convert easily between different representations of an object. Conversion is done via a set of methods, internally using bidicts. The conversion methods are usually named after a scheme, e.g. *<object><type to convert to>()*.

Example:

```
>>> registry.packetObj(obj)
obj
>>> registry.packetObj(name)
obj
>>> registry.packetObj(numid)
obj
>>> registry.packetStr(obj)
name
>>> registry.packetStr(name)
name
>>> registry.packetStr(numid)
name
>>> registry.packetInt(obj)
numid
>>> registry.packetInt(name)
numid
>>> registry.packetInt(numid)
numid
```

This is a simplified example based on `PacketRegistry`.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

p

P