Peng3dnet Documentation

Release 0.2.0

notna

May 12, 2022

Contents

Contents:

CHAPTER 1

peng3dnet - Peng3dnet Main Package

This package represents the root package of the peng3dnet networking library.

Most classes defined in submodules are also available at the package level, e.g. *peng3dnet.net.Server()* will also be available as peng3dnet.Server().

*-importing submodules and packages should generally be safe, as all modules use the __all__ variable to explicitly define their exported classes.

1.1 peng3dnet.net - Core Networking Classes

peng3dnet.net.**STRUCT_HEADER = <Struct object>**
struct.Struct instance used for rapid encoding and decoding of header data.

See also:

See *peng3dnet.constants.STRUCT_FORMAT_HEADER* for more information.

peng3dnet.net.**STRUCT_LENGTH32 = <Struct object>**
struct.Struct instance used for rapid encoding and decoding of the length prefix.

See also:

See *peng3dnet.constants.STRUCT_FORMAT_LENGTH32* for more information.

class peng3dnet.net.**Server**(*peng=None*, *addr=None*, *clientcls=None*, *cfg=None*)
Server class representing the server side of the client-server relationship.

Usually, a server will be able to serve many clients simultaneously without problems. This is achieved using the selectors standard library module, which internally uses select or similiar techniques.

If given, peng should be an instance of peng3d.peng.Peng and will be used for sending events and the configuration system. Note that without a valid peng parameter, the event system will not work and a custom config stack will be created. If a peng parameter is given, its config stack will be adapted and the event system enabled.

See also:

See *net.events.enable* and *Events used by peng3dnet* for more information on the event system.

addr, if given, should be a value parseable by *peng3dnet.util.normalize_addr_socketstyle()*. If addr is not given, first *net.server.addr* is tried, then *net.server.addr.host* and *net.server.addr.port*. If any given address is missing an explicitly specified port, *net.server.addr.port* is supplemented.

clientcls may be used to override the class used for creating new client-on-server objects. Defaults to *ClientOnServer*.

cfg may be used to override initial configuration values and should be a dictionary.

addConnType(*t*, *obj*)
Adds a connection type to the internal registry.

t should be the string name of the connection type.

obj should be an instance of a subclass of *peng3dnet.conntypes.ConnectionType()*.

Trying to register a name multiple times will cause an *AlreadyRegisteredError*.

bind()
Creates and binds the socket used for listening for new connections.

Repeated calls of this method will be ignored.

If SSL is enabled, an SSL context will be created and the socket wrapped according to the SSL settings.

See also:

See *net.ssl.enabled* for more information about the SSL configuration.

Currently, the socket will be configured to listen for up to 100 connection requests in parallel.

After binding of the socket, the event *peng3dnet:server.bind* will be sent.

close_connection(*cid*, *reason=None*)
Closes the connection to the given peer due to the optional reason.

cid should be the Client ID number.

reason may be a string describing the reason.

genCID()
Generates a client ID number.

These IDs are guaranteed to be unique to the instance that generated them.

Usually, these will be integers that simply count up and are not meant to be cryptographically secure.

initialize()
Initializes internal registries used during runtime.

Calling this method repeatedly will be ignored.

Currently, this registers packets and connection types. Additionally, the *peng3dnet:server.initialize* event is sent.

Subclasses and Mix-ins may hook into this method via definition of methods named _reg_packets_* or _reg_conntypes_* with the star being an arbitrary string. The method may not take any arguments.

interrupt()
Wakes up the main loop by sending a special message to an internal socket.

This forces the main loop to iterate once and check that the system is still running.

Also sends the *peng3dnet:server.interrupt* event.

join(*timeout=None*)
Waits for all spawned threads to finish.

If `timeout` is given, it indicates the total amount of time spent waiting.

If a thread has not been started yet, it will be skipped and not waited for.

process(*wait=False*, *timeout=None*)
Processes all packets awaiting processing.

If `wait` is true, this function will wait up to `timeout` seconds for new data to arrive.

It will then process all packets in the queue, decoding them and then calling the appropriate event handlers. This method assumes all messages are packed with msgpack.

If the connection type allows it, event handlers will be called and the *peng3dnet:server.connection.recv* event is sent.

This method returns the number of packets processed.

process_async()
Processes packets asynchronously.

Internally calls *process_forever()* in a separate daemon thread named `peng3dnet process Thread`.

process_forever()
Processes packets in a blocking manner.

Note that this method is currently not interruptable and thus uses short timeouts of about 10ms, causing a slight delay in stopping this loop.

process_single_packet(*client*)
Called when there should be enough data to process a single packet.

`client` is an instance of *ClientOnServer* representing the client.

Currently parses a single packet including length prefix and calls *receive_packet()* with the packet data.

receive_data(*data*, *cid*)
Called when new raw data has been read from a socket.

Note that the given `data` may contain only parts of a packet or even multiple packets.

`cid` is the integer ID number of the client the data was received from.

By default, the received data is stored in a buffer until enough data is available to process a packet, then *process_single_packet()* is called.

receive_packet(*data*, *cid*)
Called when a full packet has been received.

`data` is the raw packet data without length prefix.

`cid` is the integer ID number of the client.

Currently, this puts the data in a queue to be processed further by *process()*.

register_packet(*name*, *obj*, *n=None*)
Registers a new packet with the internal registry.

`name` should be a string of format `namespace:category.subcategory.name` where category may be repeated.

`obj` should be an instance of a subclass of *peng3dnet.packet.Packet()*.

`n` may be optionally used to force a packet to use a specific packet ID, otherwise one will be generated.

runAsync(*selector=<class 'selectors.EpollSelector'>*)
Runs the server main loop in a seperate thread.

`selector` may be changed to override the selector used for smart waiting.

This method does not block and should return immediately.

The newly created thread will be named `peng3dnet Server Thread` and is a daemon thread, i.e. it will not keep the program alive.

runBlocking(*selector=<class 'selectors.EpollSelector'>*)
Runs the server main loop in a blocking manner.

`selector` may be changed to override the selector used for smart waiting.

This method blocks until *`stop()`* is called.

sendEvent(*event*, *data=None*)
Helper method used to send events.

Checks if the event system is enabled, adds the `peng` and `server` data attributes and then sends it.

send_message(*ptype*, *data*, *cid*)
Sends a message to the specified peer.

`ptype` should be a valid packet type, e.g. either an ID, name or object.

`data` should be the data to send to the peer.

`cid` should be the Client ID number to send the message to.

Note that all data encoding will be done synchronously and may cause this method to not return immediately. The packet may also be encrypted and compressed, if applicable.

Additionally, the *`peng3dnet:server.connection.send`* event is sent if the connection type allows it.

shutdown(*join=True*, *timeout=0*, *reason='servershutdown'*)
Shuts down the server, disconnecting all clients.

If `join` is true, this method will block until all clients are disconnected or `timeout` seconds have passed. If `timeout` is `0`, it will be ignored.

`reason` will be used as the closing reason and transmitted to all clients.

After these messages have been sent, *`stop()`* is called and the *`peng3dnet:server.shutdown`* event is sent.

stop()
Stops the running server main loop.

Will set an internal flag, then sends the event *`peng3dnet:server.stop`* and calls *`interrupt()`* to force the close.

Note that this will not close open connections properly, use *`shutdown()`* instead.

class peng3dnet.net.**ClientOnServer**(*server*, *conn*, *addr*, *cid*)
Class representing a client on the server.

This serves mainly as a data structure for storing the state of a specific connection.

This class is not intended to be created manually.

`server` is the instance of *`Server`* that created this object.

`conn` is the socket that should be used for communication with this client.

`addr` is the address of this client.

`cid` is the unique Client ID assigned to this client.

close(*reason=None*)
Called to close the connection to this client.

This method is not an event handler, use *on_close()* instead.

on_close(*reason=None*)
Called when the connection has been closed.

`reason` is the reason sent either by the peer or passed by the caller of *Server.close_connection()*.

on_connect()
Sent once a connection has been established.

Note that at the time this method is called the handshake may not be finished, see *on_handshake_complete()* instead.

on_handshake_complete()
Called when the handshake has been completed.

The default implementation sends the *peng3dnet:server.connection.handshakecomplete* event and changes the connection state to *STATE_ACTIVE*.

May be overridden by subclasses.

on_receive(*ptype*, *msg*)
Called when a packet has been received.

`ptype` will be an integer or string representing the packet type.

`msg` will be a Python object decoded from the raw message via messagepack.

on_send(*ptype*, *msg*)
Called when a packet has been sent.

`ptype` will be the packet type in either string, integer or object form.

`msg` will be the Python object that has been encoded and sent via messagepack.

class peng3dnet.net.**Client**(*peng=None*, *addr=None*, *cfg=None*, *conntype='classic'*)
Client class representing the client side of the client-server relationship.

A client can only be connected to a single server during its lifetime, recycling of client instances is not supported.

If given, `peng` should be an instance of `peng3d.peng.Peng` and will be used for sending events and the configuration system. Note that without a valid `peng` parameter, the event system will not work and a custom config stack will be created. If a `peng` parameter is given, its config stack will be adapted and the event system enabled.

See also:

See *net.events.enable* and *Events used by peng3dnet* for more information on the event system.

`addr`, if given, should be a value parseable by *peng3dnet.util.normalize_addr_socketstyle()*. If `addr` is not given, first *net.client.addr* is tried, then *net.client.addr.host* and *net.client.addr.port*. If any given address is missing an explicitly specified port, *net.client.addr.port* is supplemented.

`cfg` may be used to override initial configuration values and should be a dictionary.

Optionally, the connection type may be specified via `conntype`, which may be set to a string identifying the type of the connection. This should usually be *CONNTYPE_CLASSIC* or one of the other `CONNTYPE_*` constants. Note that the connection type specified must also be registered via *addConnType()*, except for the built-in connection types.

addConnType(*t*, *obj*)
Adds a connection type to the internal registry.

`t` should be the string name of the connection type.

`obj` should be an instance of a subclass of *peng3dnet.conntypes.ConnectionType()*.

Trying to register a name multiple times will cause an *AlreadyRegisteredError*.

close(*reason=None*)
Called to close the connection to the server.

This method is not an event handler, use *on_close()* instead.

Also sends the event *peng3dnet:client.close*.

close_connection(*cid=None*, *reason=None*)
Closes the connection to the server due to the optional reason.

`cid` is a dummy value used for compatibility with server applications.

`reason` may be a string describing the reason.

connect()
Connects the client with a server.

Note that the server must have been specified before calling this method via either the `addr` argument to the initializer or any of the *net.client.addr* config values.

Repeated calls of this method will be ignored.

If SSL is enabled, this method will also initialize the SSL Context and load the certificates.

After the connection has been made, the `peng3dnet.client.connect` event is sent.

initialize()
Initializes internal registries used during runtime.

Calling this method repeatedly will be ignored.

Currently, this registers packets and connection types. Additionally, the *peng3dnet:client.initialize* event is sent.

Subclasses and Mix-ins may hook into this method via definition of methods named `_reg_packets_*` or `_reg_conntypes_*` with the star being an arbitrary string. The methods may not take any arguments.

interrupt()
Wakes up the main loop by sending a special message to an internal socket.

This forces the main loop to iterate once and check that the system is still running.

Also sends the *peng3dnet:client.interrupt* event.

join(*timeout=None*)
Waits for all spawned threads to finish.

If `timeout` is given, it indicates the total amount of time spent waiting.

If a thread has not been started yet, it will be skipped and not waited for.

on_close(*reason=None*)
Event handler called during connection shutdown.

It may or may not be possible to send data over the connection within this method, depending on various factors.

on_connect()
Event handler called once a connection has been established.

Note that usually the handshake will still be in progress while this method is called.

See also:

See the *on_handshake_complete()* event handler for a better indicator when data can be sent.

on_handshake_complete()
Callback called once the handshake has been completed.

The default implementation sends the *peng3dnet:client.handshakecomplete* event and sets the connection state to *STATE_ACTIVE*.

on_receive(*ptype*, *msg*)
Event handler called if data is received from the peer.

on_send(*ptype*, *msg*)
Event handler called if data is sent to the peer.

process(*wait=False*, *timeout=None*)
Processes all packets awaiting processing.

If `wait` is true, this function will wait up to `timeout` seconds for new data to arrive.

It will then process all packets in the queue, decoding them and then calling the appropriate event handlers. This method assumes all messages are packed with msgpack.

If the connection type allows it, event handlers will be called and the *peng3dnet:client.recv* event is sent.

This method returns the number of packets processed.

process_async()
Processes packets asynchronously.

Internally calls *process_forever()* in a separate daemon thread named `peng3dnet process Thread`.

process_forever()
Processes packets in a blocking manner.

Note that this method is currently not interruptable and thus uses short timeouts of about 10ms, causing a slight delay in stopping this loop.

process_single_packet(*client=None*)
Called when there should be enough data to process a single packet.

`client` is a dummy value used for compatibilizy with server applications.

Currently parses a single packet including length prefix and calls *receive_packet()* with the packet data.

pump_write_buffer()
Tries to send as much of the data in the internal buffer as possible.

Note that depending on various factors, not all data may be sent at once. It is possible that sent data will be fragmented at arbitrary points.

If an exception occurs while sending the data, it will be ignored and the error printed to the console.

receive_data(*data*, *cid=None*)
Called when new raw data has been read from the socket.

Note that the given `data` may contain only parts of a packet or even multiple packets.

`cid` is a dummy value used for compatibility with server applications.

By default, the received data is stored in a buffer until enough data is available to process a packet, then *process_single_packet()* is called.

receive_packet(*data*, *cid=None*)
Called when a full packet has been received.

`data` is the raw packet data without length prefix.

`cid` is a dummy value used for compatibility with server applications.

Currently, this puts the data in a queue to be processed further by *process()*.

register_packet(*name*, *obj*, *n=None*)
Registers a new packet with the internal registry.

`name` should be a string of format `namespace:category.subcategory.name` where category may be repeated.

`obj` should be an instance of a subclass of *peng3dnet.packet.Packet()*.

`n` may be optionally used to force a packet to use a specific packet ID, otherwise one will be generated.

runAsync(*selector=<class 'selectors.EpollSelector'>*)
Runs the client main loop in a seperate thread.

`selector` may be changed to override the selector used for smart waiting.

This method does not block and should return immediately.

The newly created thread will be named `peng3dnet Client Thread` and is a daemon thread, i.e. it will not keep the program alive.

runBlocking(*selector=<class 'selectors.EpollSelector'>*)
Runs the client main loop in a blocking manner.

`selector` may be changed to override the selector used for smart waiting.

This method blocks until *stop()* is called.

sendEvent(*event*, *data*)
Helper method used to send events.

Checks if the event system is enabled, adds the `peng` and `client` data attributes and then sends it.

send_message(*ptype*, *data*, *cid=None*)
Sends a message to the server.

`ptype` should be a valid packet type, e.g. either an ID, name or object.

`data` should be the data to send to the server.

`cid` will be ignored, available for compatibility with server applications.

Note that all data encoding will be done synchronously and may cause this method to not return immediately. The packet may also be encrypted and compressed, if applicable.

Additionally, the *peng3dnet:client.send* event is sent if the connection type allows it.

stop()
Stops the running client main loop.

Will set an internal flag, then sends the event *peng3dnet:client.stop* and calls *interrupt()* to force the close.

Note that this will not close open connections properly, call *close_connection()* before calling this method.

wait_for_close(*timeout=None*)
Waits up to `timeout` seconds for the connection to close.

Returns immediately if the connection is already closed.

wait_for_connection(*timeout=None*)
Waits up to `timeout` seconds for the connection to be established.

Returns immediately if there is an active connection.

1.2 peng3dnet.constants - Constants

This module is designed to be imported via `from peng3dnet.constants import *` without any side-effects.

peng3dnet.constants.**MAX_PACKETLENGTH = 4294967295**
Constant equal to the maximum packet length representable by the length prefix in bytes.

Currently equals roughly 4.3GB or `2**32-1` Bytes.

peng3dnet.constants.**STRUCT_FORMAT_LENGTH32 = '!I'**
Format of the struct used for the length prefix.

This struct should be able to store a single integer value.

Note that changing this format string will break compatibility with non-modified peers.

For performance reasons, *peng3dnet.net.STRUCT_LENGTH32* should be used during runtime instead.

peng3dnet.constants.**STRUCT_FORMAT_HEADER = '!IH'**
Format of the struct used for the packet header.

This struct should be able to store a packet id encoded as an integer and a bitfield containing flags.

Note that changing this format string will break compatibility with non-modified peers.

For performance reasons, *peng3dnet.net.STRUCT_HEADER* should be used during runtime instead.

peng3dnet.constants.**STATE_INIT = 0**
Default connection state of every new connection.

peng3dnet.constants.**STATE_HANDSHAKE_WAIT1 = 1**
Connection state symbolizing the peer has to wait until its peer sends the required packet.

This connection state is part of the internal handshake and should not be used manually.

peng3dnet.constants.**STATE_HANDSHAKE_WAIT2 = 2**
Currently unused connection state.

peng3dnet.constants.**STATE_WAITTYPE = 3**
Active connection state if the server is waiting for the client to send the connection type.

This connection state is part of the internal handshake and should not be used manually.

peng3dnet.constants.**STATE_HELLOWAIT = 4**
Connection state used by the client to wait for the server to send a *HelloPacket*.

This connection state is part of the internal handshake and should not be used manually.

peng3dnet.constants.**STATE_LOGGEDIN = 65**
Generic state indicating an active connection and successful authentication.

peng3dnet.constants.**STATE_ACTIVE = 64**
Generic state indicating an active connection and successful handshake.

peng3dnet.constants.**STATE_CLOSED = 128**
Internal state indicating a closed connection that may be removed completely.

peng3dnet.constants.**MODE_NOTSET = 0**
Placeholder mode used to indicate that the mode has not yet been set.

peng3dnet.constants.**MODE_CLOSED = 1**
Internal mode used to indicate that the connection is not active.

peng3dnet.constants.**MODE_PING = 2**
Old internal mode used to indicate that a ping request is being made.

peng3dnet.constants.**MODE_PLAY = 3**
Generic mode used to indicate that the connection is in play mode.

The exact definition of this mode is up to the application.

peng3dnet.constants.**MODE_CHAT = 4**
Generic mode used to indicate that the connection is in chat mode.

peng3dnet.constants.**CONNTYPE_NOTSET**
Placeholder connection type.

Note that this connection type will cause errors if used, use *CONNTYPE_CLASSIC* instead.

peng3dnet.constants.**CONNTYPE_CLASSIC**
Classic connection type.

See also:

See *peng3dnet.conntypes.ClassicConnectionType* for more information.

peng3dnet.constants.**CONNTYPE_PING = 'ping'**
Ping request connection type.

peng3dnet.constants.**FLAG_COMPRESSED = 1**
Flag bit set if the packet is compressed.

peng3dnet.constants.**FLAG_ENCRYPTED_AES = 2**
Flag bit set if the packet is AES-Encrypted.

Note that this flag is currently not implemented.

peng3dnet.constants.**SIDE_CLIENT = 0**
Constant used to indicate the client side of the server-client relationship.

peng3dnet.constants.**SIDE_SERVER = 1**
Constant used to indicate the server side of the server-client relationship.

peng3dnet.constants.**SSLSEC_NONE = 0**
SSL Security level indicating that there is no SSL tunneling active.

Default SSL Security level.

peng3dnet.constants.**SSLSEC_WRAPPED = 1**
SSL Security level indicating that the socket has been wrapped in a SSLSocket.

Note that this usually isn't more secure compared to *SSLSEC_NONE*.

peng3dnet.constants.**SSLSEC_ENCRYPTED = 2**
SSL Security level indicating that the SSL Tunnel uses encryption.

Only encryption enabled by the default in the standard library module `ssl` is regarded as secure.

peng3dnet.constants.**SSLSEC_SERVERAUTH = 3**
SSL Security level indicating that the server certificate is authentic and valid.

This includes hostname verification.

peng3dnet.constants.**SSLSEC_BOTHAUTH = 4**
SSL Security level indicating that both server and client certificate are authentic and valid.

This includes hostname verification.

peng3dnet.constants.**DEFAULT_CONFIG**
Default configuration values.

This dictionary is used to look up default config values.

See also:

See *Configuration Options for peng3dnet* for a list of all config options.

1.3 peng3dnet.conntypes - Connection Types

class peng3dnet.conntypes.**ConnectionType**(*peer*)
Class representing a Connection Type implementation.

Connection Types are identified by their name, usually a lowercase string.

See *Client()* for how to specify the connection type used.

To use a custom connection type, it must be registered on both server and client via *Server.addConnType()* or *Client.addConnType()*, respectively.

`peer` is an instance of either *Client()* or *Server()*.

Note that a single instance of this class will be shared between all connections of the type implemented by this class. Distinguishing single connections is possible via the `cid` parameter given to most methods. On the client side, this parameter will always be `None`.

init(*cid*)
Called when the *SetTypePacket()* is received on the server side, or sent on the client side.

Detecting which side of the connection is managed can be done by checking the `cid` parameter, if it is `None`, the client side is represented, else it represents the ID of the connected client.

receive(*msg*, *pid*, *flags*, *cid*)
Called whenever a packet is received via connection of the type represented by this class.

`msg` is the already decoded message or payload.

`pid` is the ID of the packet type.

`flags` is the flags portion of the header, containing a bitfield with various internal flags.

`cid` is the ID of the connected peer, if it is `None`, the peer is a server.

If the return value of this method equals to `True`, further processing of the packet will be prevented.

send(*data*, *pid*, *cid*)
Called whenever a packet has been sent via a connection of the type represented by this class.

`data` is the fully encoded data that has been sent.

`pid` is the packet type, as received by either *Server.send_message()* or *Client.send_message()*.

`cid` is the ID of the connected peer, it if is `None`, the peer is a server.

If the return value of this method equals to `True`, no further event handlers will be called.

class peng3dnet.conntypes.**ClassicConnectionType**(*peer*)
Classic Connection Type representing a typical connection.

Currently adds no further processing to packets and starts a handshake by sending a *HandshakePacket()* from the server to the client.

The handshake allows the client to copy the registry of the server, preventing bugs with mismatching packet IDs.

1.4 peng3dnet.packet - Packet Classes

class peng3dnet.packet.**Packet**(*reg*, *peer*, *obj=None*)
Class representing a packet type.

`reg` is an instance of *PacketRegistry* shared between all packet types registered with it.

`peer` is the peer this packet type belongs to.

Todo: Find out what `obj` does... Seems to be an unused artifact from a test.

_receive(*msg*, *cid=None*)
Internal handler called whenever a packet of this type is received.

By default, this method passes its arguments through to *receive()*.

May be overridden by subclasses to prevent further processing.

_send(*msg*, *cid=None*)
Internal handler called whenever a packet of this type has been sent.

By default, this method passes its arguments through to *send()*.

May be overridden by subclasses to prevent further processing.

receive(*msg*, *cid=None*)
Event handler called when a packet of this type is received.

Note that this method is usually called by *_receive()*, which may also decide to silently ignore a packet.

`msg` is the fully decoded message or payload, usually a dictionary.

`cid` is the ID of the peer that sent the message. If this is `None`, the packet was received on the client side, else it equals the client ID of the client.

send(*msg*, *cid=None*)
Event handler called when a packet of this type has been sent.

Note that this method is usually called by *_send()*, which may also decide to silently ignore a packet.

`msg` is the already encoded message.

`cid` is the ID of the peer that should receive the message. If this is `None`, the packet was sent on the client side, else it equals the client ID of the recipient.

class peng3dnet.packet.**SmartPacket**(*reg*, *peer*, *obj=None*)
Smart packet type allowing for various assertions.

This class is not intended to be used directly, use subclasses and override the class attributes for customization.

This class overrides the `_receive()` and `_send()` methods to check that the connection is in a valid state for this packet.

conntype = None
Declares the required connection type for this packet to be sent or retrieved.

If this is `None`, this check will be skipped.

Else, if the specified value does not match the actual value, the action specified in *invalid_action* is executed.

invalid_action = 'ignore'
Specifies what to do if one of the conditions specified is not fulfilled.

Can be either `ignore` or `close`. A value of `ignore` causes this packet to be ignored. Note that due to technical reasons, the packet may still be processed by other mechanisms. In contrast, a value of `close` will cause the connection to be closed with the reason `smartpacketinvalid`.

If an invalid value is used, an *InvalidSmartPacketActionError* will be raised and the packet ignored.

min_sslsec_level = 0
Declares the minimum level of SSL encryption required for this packet to be sent.

This is not checked on receival of a message, only during sending of the message.

Possible values are any of the `SSLSEC_*` constants.

If the actual SSL security level is lower than the required level, the action specified in *invalid_action* is executed.

mode = None
Specifies the connection mode required for this packet to be sent or received.

If this is `None`, this check will be skipped.

Else, if the specified connection mode does not exactly match the actual connection mode, the action specified in *invalid_action* is executed.

side = None
Specifies the side of the connection this packet can be received.

If this is `None`, it can be received on both ends of a connection. If this is *SIDE_SERVER* or *SIDE_CLIENT*, it can be received on the server or client side, respectively.

Note that packets that can only be received on the client side can only be sent from the server side, and vice versa.

In the case that this condition does not match, the action specified in *invalid_action* is executed.

state = 64
Specifies the connection state required for the packet, both on send and receive.

If the state does not match exactly, the action specified in *invalid_action* is executed.

class peng3dnet.packet.**PrintPacket**(*reg*, *peer*, *obj=None*)
Simple dummy packet class that prints any message it receives.

If a packet is received on the server, the client ID of the client is also printed.

1.5 peng3dnet.packet.internal - Internal Packet Classes

This module contains various internal packets not intended for direct use.

These packets will be registered by the default implementations of client and server classes with names starting with `peng3dnet:`. Any other packets whose name starts with `peng3dnet:` may be processed incorrectly.

Additionally, packets within this module usually use reserved and static packet IDs below 16.

1.5.1 Handshake

Note that if a custom connection type is used, any steps after step 3. may be left out.

1. Server sends a *HelloPacket* with version information
2. Client responds with *SetTypePacket* containing connection type
3. Server stores connection type and sends *HandshakePacket* containing version and registry
4. Client updates own registry based on packet and sends *HandshakeAcceptPacket*
5. Server receives packet and calls event handler to signal a successful handshake

1.5.2 Connection shutdown

Note that this only applies to clean shutdowns caused by *close_connection()* or its client-side equivalent.

1. `close_connection()` is called
2. *CloseConnectionPacket* is sent to peer and internal flag is set
3. After packet has been fully sent, event handlers are called
4. Peer receives packet and calls event handlers
5. Server cleans up internal data structures

class peng3dnet.packet.internal.**HelloPacket**(*reg*, *peer*, *obj=None*)
Internal packet sent by the server to initialize a handshake.

This is usually the first packet transmitted by every connection. It contains version information for the client to check.

If the client does not support the given protocol version, the connection must be aborted with the reason `protoversionmismatch`.

class peng3dnet.packet.internal.**SetTypePacket**(*reg*, *peer*, *obj=None*)
Internal packet sent by the client to indicate the connection type.

If the server does not recognize the connection type, the connection must be aborted with the reason `unknownconntype`.

If no connection type is supplied, `classic` is substituted.

class peng3dnet.packet.internal.**HandshakePacket**(*reg*, *peer*, *obj=None*)
Internal packet sent by the server to synchronize the registry.

Additionally, the version information is sent and checked.

If the *net.registry.autosync* config value is true, the registry sent by the server will be adapted to the client.

Note that only IDs are synced to names, objects will not be affected.

class peng3dnet.packet.internal.**HandshakeAcceptPacket**(*reg*, *peer*, *obj=None*)
Internal packet sent by the client to indicate a successful handshake.

Once this packet has been sent or received, the connection is established and can be used.

class peng3dnet.packet.internal.**CloseConnectionPacket**(*reg*, *peer*, *obj=None*)
Internal packet to be sent to indicate that a connection is about to be closed.

Usually includes a reason in the `reason` field.

This packet can be sent at any time, regardless of connection state or type.

1.6 peng3dnet.ext - Extensions

1.7 peng3dnet.ext.ping - One-off data retrieval protocol extension

The ping extension was designed to allow a client to check on various values the server provides without much effort. It allows the client to check on metrics such as latency and additional values able to be customized by the server.

This extension is named after the UNIX **ping** utility, though it can do much more than just check on availability and latency.

class peng3dnet.ext.ping.**PingConnectionType**(*peer*)
Connection type to be used by ping connections.

This connection type prevents any synchronization of the registry to allow clients only supporting a subset of the peng3dnet protocol to still ping a server.

Additonally, conventional processing of packets will be disabled by this connection type, making it uneccessary to register packets with the client or server.

getPingData(*msg*, *cid=None*)
Overridable method to create a ping response.

`msg` is the ping query, as received from the client.

`cid` is the ID of the client.

Called only on the server side.

init(*cid*)
Called whenever a new ping connection is established.

On the client, this calls `PingableClientMixin._ping()` and updates the connection state, while on the server only the connection state is updated.

receive(*msg*, *pid*, *flags*, *cid*)
Called whenever a packet is received via this connection type.

Handles any ping requests and pong answers and always returns `True` to skip any further processing.

send(*msg*, *pid*, *cid*)
Called whenever a packet is sent via this connection type.

class peng3dnet.ext.ping.**PingableClientMixin**
Mixin for *Client* classes enabling support for pinging the server.

Currently automatically adds the `ping` connection type.

setPingData(*d*)
Sets the data to add to any ping responses.

Repeated calls of this method will overwrite previous data.

wait_for_pong(*timeout=None*)
Waits up to `timeout` seconds for a ping response to arrive.

If a response has already been received, this method returns immediately.

If the ping was successful, the received message is returned.

class peng3dnet.ext.ping.**PingableServerMixin**
Mixin for *Server* classes enabling support for pinging the server.

Currently automatically adds the `ping` connection type.

getPingData(*msg*, *cid*)
Overrideable method called to extend the default dictionary returned upon a ping request.

May be overriden to add dynamic data like user count or similiar information.

`msg` is the original message as received from the client.

`cid` is the client ID that made this request.

pingdata = {}
Overrideable dictionary used to extend the default dictionary returned upon a ping request.

May be overriden to add static data like server name or similiar information.

peng3dnet.ext.ping.**WRITEBACK = True**
Constant allowing to configure if the received ping message should be transmitted back to the client.

If there should ever be any security concerns regarding that feature, this flag can simply be flipped.

peng3dnet.ext.ping.**pingServer**(*peng=None*, *addr=None*, *cfg=None*, *data=None*, *clientcls=<class 'peng3dnet.ext.ping._PingClient'>*, *timeout=10.0*)
Pings the specified server.

Internally, this creates a client that supports pinging and listens for any data received back.

`peng` may be optionally used to replace the argument of the same name to *Client()*.

`addr` specifies the address of the server to ping.

`cfg` may be used to override the configuration for the client, e.g. SSL settings.

`data` is the data sent to the server. Note that the `time` key will be overridden for measuring the latency.

`clientcls` may be used to override the client class used.

`timeout` is maximum amount of time to wait for a response.

The data returned will be the data received from the server, except for additional information that has been added. Currently, the `recvtime` key contains the timestamp that the response was received and the `delay` key contains the total roundtrip time in seconds.

1.8 peng3dnet.registry - Smart Registry Class

class peng3dnet.registry.**BaseRegistry**(*objtype=None*)
Basic registry class.

Supports smart conversions between the integer, string and generic object representation of a registered entity.

Optionally allows for automatic and threadsafe integer ID generation.

Requires `bidict` to be installed and available.

`objtype` may be used to override the class attribute of the same name per instance.

Instances of this class also support dictionary-style access to their data, e.g. `reg[val]` will always return the object representation of the value, see *getObj()* for details.

deleteObj(*obj*)
Removes an object from the internal registry.

`obj` may be any of the three representations of an object.

getAll(*obj*)
Returns a three-tuple of form `(getObj(obj),getID(obj),getName(obj))`.

getID(*obj*)
Converts the given value to its integer representation.

This method accepts either strings, integers or objects of type *objtype*.

`getInt()` may be used as an alias to this method.

getName(*obj*)
Converts the given value to its string representation.

This method accepts either strings, integers or objects of type *objtype*.

`getStr()` may be used as an alias to this method.

getNewID()
Generates a new ID.

Currently, all IDs are increasing from a fixed starting point, by default `64`.

getObj(*obj*)
Converts the given value to its object representation.

This method accepts either strings, integers or objects of type *objtype*.

objtype
alias of `builtins.object`

register(*obj*, *name*, *n=None*)
Registers a relation between an object, its string representation and its integer representation.

If `n` is not given, *getNewID()* will be used to generate it.

Trying to register an already registered object may cause various kinds of corruptions on internal storages.

Trying to register an object that is not of the type specified in *objtype* will result in an `TypeError`.

registerObject(*obj*, *n=None*)
Same as *register()*, but extracts the string representation from the object's `name` attribute.

class peng3dnet.registry.**PacketRegistry**(*objtype=None*)
Subclass of *BaseRegistry* customized for storing *Packet* instances and instances of subclasses.

objtype
alias of *peng3dnet.packet.Packet*

1.9 peng3dnet.util - Utility functions

peng3dnet.util.**normalize_addr_formatted**(*addr*)
Normalizes the given address to a string like `127.0.0.1`.

This method is currently not implemented.

peng3dnet.util.**normalize_addr_socketstyle**(*addr*, *default_port=8080*)
Normalizes the given address to a 2-tuple as accepted by the `socket` module.

Currently accepts a 2-tuple and IPv4 addresses in string format.

If the address does not contain a port, the `default_port` will be used.

Note that this function will pass through any exceptions raised by parsing functions it calls.

peng3dnet.util.**parse_address**(*addr*, *default_port=8080*)
Parses an IP Address into a tuple of `(addr,port)`.

If the address does not contain an explicitly specified port, the value given with `default_port` is used.

Note that currently only IPv4 addresses are supported, but IPv6 support may be added in the future. If an IPv6 Address is detected, a *UnsupportedAddressError* will be raised.

Additionally, the port returned is checked for plausibility, e.g. an integer in range 0-65535. If the port is invalid in any way, a *InvalidPortError* will be raised.

1.10 peng3dnet.errors - Exception Classes

This module contains the various exception classes used by peng3dnet.

Most methods and functions that use these exceptions will have a link to the appropriate exception in their documentation.

exception peng3dnet.errors.**InvalidAddressError**
Indicates that a given address is not valid and thus cannot be used.

exception peng3dnet.errors.**InvalidPortError**
Indicates that the port supplied or parsed is not valid.

exception peng3dnet.errors.**InvalidHostError**
Indicates that the host supplied or parsed is not valid or applicable.

exception peng3dnet.errors.**UnsupportedAddressError**
Indicates that the address supplied is not supported, but may still be valid.

exception peng3dnet.errors.**InvalidSmartPacketActionError**
Raised if the `invalid_action` of a *SmartPacket* is not valid.

exception peng3dnet.errors.**TimedOutError**
Indicates that some action has timed out, this includes connections, requests and any other applicable action.

exception peng3dnet.errors.**FailedPingError**
Indicates that a ping request has failed, usually due to a timeout.

exception peng3dnet.errors.**RegistryError**
Indicates that a registry has encountered an error.

exception peng3dnet.errors.**AlreadyRegisteredError**
Indicates that the object given has already been registered.

1.11 peng3dnet.version - Version Information

peng3dnet.version.**VERSION = '0.2.0'**
Full version number of this package.

Used to display the version in the title of the documentation and in `setup.py`.

See also:

See Semantic Versioning for more information on the scheme used by this application.

peng3dnet.version.**RELEASE = '0.2.0'**
Full version number of this package without trailing meta-data.

See *VERSION* for more information.

peng3dnet.version.**PROTOVERSION = 2**
Version of the protocol used internally.

This value is shared during the handshake of a connection.

Two versions of this library that do not share the same protocol versions are probably not compatible.

CHAPTER 2

Configuration Options for peng3dnet

These configuration options can be used to manipulate various features of peng3dnet. The default values for these config options have been set to allow to not need any config changes for most basic applications.

Config values can be changed by either passing a dictionary with overiddes as the `cfg` argument to *`Client()`* or *`Server()`*, or by accessing the `cfg` attribute of an instance of the classes mentioned above.

Note that all config values added by peng3dnet have the `net.` prefix to prevent confusion.

Default config values are mostly documented within the docs themselves or can be found in *`peng3dnet.constants.DEFAULT_CONFIG`*.

2.1 `net.server.*` - Generic Server-side config options

All non feature-specific server-side config options will use the ‘‘net.server.*‘ prefix.

`net.server.addr`
`net.server.addr.host`
`net.server.addr.port`

These config options specify the host and port the server will listen on.

See also:

See *`peng3dnet.net.Server`* for more information on how these config options work.

`net.server.addr` defaults to `None`, while *`net.server.addr.host`* and :confval:net.server.addr.port‘ default to `0.0.0.0` and `8080`, respectively.

2.2 `net.client.*` - Generic Client-side config options

All non feature-specific client-side config options will use the ‘‘net.client.*‘ prefix.

`net.client.addr`
`net.client.addr.host`

net.client.addr.port
These config options specify the host and port the client will connect to.

See also:

See *peng3dnet.net.Client* for more information on how these config options work.

net.client.addr defaults to `None`, while *net.client.addr.host* and *net.client.addr.port* default to `localhost` and `8080`, respectively.

2.3 net.compress.* - Compression settings

These config options apply to the per-packet compression. It is recommended that client and server use the same config options for better compatibility, but it is not strictly necessary.

net.compress.enabled
Determines whether or not the compression module is activated.

If this config options is `False`, all outgoing packets will be uncompressed, but incoming compressed packets should still be processed appropriately assuming the required modules are available.

This config option defaults to `True`.

net.compress.threshold
Determines the compression threshold in bytes.

All packets with a size greater than this config option will be compressed.

Defaults to `8192`, or 8 Kib.

net.compress.level
The `zlib` compression level to use when compressing packets.

See also:

Please see `zlib.compress()` for more information about compression levels.

This config option defaults to `6`

2.4 net.encrypt.* - Encryption settings

This module is currently not implemented.

net.encrypt.enabled
Determines whether or not the encryption module is activated.

This module is currently not implemented, changing this option should still not be done.

Defaults to `False`.

2.5 net.ssl.* - SSL settings

These config options affect the SSL configuration used by both server and client.

Note that currently the SSL module is very buggy and thus disabled by default. It should not be used for any serious applications.

The default config values are configured for maximum security, this means that servers must always have certificates that will be verified.

net.ssl.enabled
Determines whether or not the SSL module is activated.

It is necessary that both sides have SSL enabled, or the connection will fail with an undefined error.

This config option defaults to `False`.

net.ssl.force
Used to configure if loading the `ssl` module is required.

If `True`, an error will be raised if the ssl module is not available. If `False` and the ssl module is not available, it will simply not be loaded.

This config option defaults to `True`.

net.ssl.cafile
net.ssl.server.force_verify
net.ssl.server.certfile
net.ssl.server.keyfile
net.ssl.client.check_hostname
net.ssl.client.force_verify
These options mostly do what they are named after, further documentation is currently not provided due to frequent design changes regarding them.

For default values see *peng3dnet.constants.DEFAULT_CONFIG*.

2.6 net.events.* - Event settings

net.events.enable
Determines whether or not events will be sent.

If `auto` is used, events will be sent only if a `peng` instance has been passed to the client or server.

Defaults to `auto`.

2.7 net.debug.* - Internal Debug Flags

net.debug.print.recv
net.debug.print.send
net.debug.print.connect
net.debug.print.close
Flags that determine whether or not the specific status messsages will be printed out.

These flags are temporary until a proper logging system has been established.

All of these options default to `False`.

2.8 net.registry.* - Registry Configuration

net.registry.autosync
Determines whether or not the registry will be automatically synced between server and client.

See *HandshakePacket()* for more information about the auto-sync.

This config option defaults to `True`.

`net.registry.missingpacketaction`
Configures what action is taken if during registry synchronization the client and server do not have the same packets.

Currently possible values are `closeconnection` which closes the connection and `ignore` which simply ignores the mismatch but may cause issues later.

This config option defaults to `closeconnection`.

CHAPTER 3

Events used by peng3dnet

The event system of peng3d is used by peng3dnet to allow peng3d apps to easily integrate with peng3dnet.

See also:

See the peng3d docs about the Peng3d Event System for more information about the peng3d event system.

3.1 Server-Side Events

These events are sent exclusively by the *Server()* class.

Unless otherwise noted, they always contain the peng and server keys in their data dictionary.

peng will be an instance of peng3d.peng.Peng() or None, as supplied to the constructor.

server will be the instance of *Server()* that sent the event.

peng3dnet:server.initialize
: Sent once the server has been initialized via *peng3dnet.net.Server.initialize()*.

  This event has no additional data attached to it.

peng3dnet:server.bind
: Sent once the server-side socket has been bound to a specific address via *peng3dnet.net.Server.bind()*.

  This event has the additional data key addr set to a 2-tuple of format (host,port).

peng3dnet:server.start
: Sent when the server main loop is about to start.

  This event has no additional data attached to it.

peng3dnet:server.stop
: Sent whenever the *peng3dnet.net.Server.stop()* method is called.

  The data key reason will be set to method to indicate that the stop has been caused by calling the stop method. In the future, other reasons may be introduced.

peng3dnet:server.interrupt
Sent whenever an interrupt has been sent via *peng3dnet.net.Server.interrupt()*.

Note that the processing of the interrupt may be delayed due to various factors.

This event has no additional data attached to it.

peng3dnet:server.shutdown
Sent once the server has been shutdown by calling the *peng3dnet.net.Server.shutdown()* method.

All arguments passed to *peng3dnet.net.Server.shutdown()* will be present in the data attached to this event.

3.1.1 Connection-specific Events

These events are specific to a connection and will often be triggered very frequently. This makes it important that event handlers subscribing to events in this subsection are high-performant, as they may significantly impact overall performance.

peng3dnet:server.connection.accept
Sent whenever a new connection is established.

Note that the connection may not be able to transmit data, see *peng3dnet:server.connection.handshakecomplete* for more information.

This event has the following data attached to it:

`sock` is the actual TCP socket of the connection.

`addr` is the remote address that has connected to the server.

`client` is an instance of *ClientOnServer()* used to represent the client.

`cid` is the numerical ID of the client.

peng3dnet:server.connection.handshakecomplete
Sent whenever a handshake with a client has been completed.

Usually, this means that the client in question is now able to both send and receive packets.

The data key `client` will be set to the instance of *ClientOnServer()* that represents the client.

peng3dnet:server.connection.send
Sent whenever a packet has been sent over a connection.

Note that this event is only triggered if the connection type allows it.

Additional data:

`client` is the instance of *ClientOnServer()* representing the target client.

`pid` is the packet type, as given to *peng3dnet.net.Server.send_message()*.

`data` is the encoded packet, including header and length-prefix.

peng3dnet:server.connection.recv
Sent whenever a packet has been received from a connection.

Note that this event is only triggered if the connection type allows it.

Additional data:

`client` is the instance of *ClientOnServer()* representing the sender of the packet.

`pid` is the packet type, as an integer.

`msg` is the fully decoded message. Usually, this will be a dictionary, but other types are possible.

peng3dnet:server.connection.close
Sent whenever a connection has been closed.

This event should only be sent once per connection, though this is not guaranteed.

After this event has been sent, the connection will be cleant up, meaning it is no longer available to send or receive.

Additional data:

`client` is the instance of *ClientOnServer()* representing the connection to be closed.

`reason` is a string or `None` representing the reason this connection has been closed. Note that these reasons are not standardized and may change at any point in time.

3.2 Client-Side Events

These events are sent exclusively by the *Client()* class.

Unless otherwise noted, they always contain the `peng` and `client` keys in their data dictionary.

`peng` will be an instance of `peng3d.peng.Peng()` or `None`, as supplied to the constructor.

`client` will be the instance of *Client()* that sent the event.

peng3dnet:client.initialize
Sent once the client has been initialized by calling *peng3dnet.net.Client.initialize()*.

Will be sent exactly once per client.

This event has no additional data attached to it.

peng3dnet:client.connect
Sent once the client has been connected to a server via *peng3dnet.net.Client.connect()*.

Note that this event only signals that the underlying connection has been established, the SSL tunnel and Handshake may not yet be working.

Will be sent exactly once per client.

Additional data:

`addr` will be the address of the server in the format `(host,port)`

`sock` will be the socket itself used to communicate with the server.

peng3dnet:client.start
Sent once the client has been started via *peng3dnet.net.Client.runBlocking()*.

Note that this event will be sent once per instance of *Client()*.

This event has no additional data attached to it.

peng3dnet:client.stop
Sent whenever the *peng3dnet.net.Client.stop()* method is called.

The data key `reason` will be set to `method` to indicate that the stop has been caused by calling the stop method. In the future, other reasons may be introduced.

peng3dnet:client.interrupt
Sent whenever an interrupt has been issued by *peng3dnet.net.Client.interrupt()*.

Note that the actual processing of the interrupt may be delayed by an arbitrary time.

peng3dnet:client.handshakecomplete
Sent once the handshake has been completed.

Note that some connection types may not trigger this event.

This event has no additional data attached to it.

peng3dnet:client.recv
Sent whenever a packet has been received from the server.

Note that this event is only triggered if the connection type allows it.

Additional data:

`pid` is the packet type, as an integer.

`msg` is the fully decoded message. Usually, this will be a dictionary, but other types are possible.

peng3dnet:client.send
Sent whenever a packet is about to be sent to the server.

Note that this event is only triggered if the connection type allows it.

Additional data:

`pid` is the packet type, as given to *peng3dnet.net.Client.send_message()*.

`data` is the raw packet data before encoding.

peng3dnet:client.close
Sent once the connection to the server has been closed.

This event will usually be sent only once per client.

The `reason` data key will be set to the reason as either a string or `None`.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

F

G

H

I

J

M

N

O

P

R

S

T

U

V

W