osc4py3 Documentation

Release 1.0.8

Laurent Pointal

Aug 07, 2018

Contents

- Module documentation (on Read The Docs)
- Subversion repository & bug tracking (on french academic SourceSup site).
- Developer page

Osc4py3 is an implementation of Open Sound Control (OSC) message transport protocol within a Python3 package.

It manages different sides of OSC in possibly different contexts:

- encoding/decoding of OSC message packets (including bundles)
- routing of incoming messages based on selector regexps or globbing
- timed messages with possible delay period
- named client/server for sending/subscribing (light coupling)
- different scheduling models (single process, totally multithread, only multithread for communications)
- extra processing of packets (hack points to encrypt/decrypt, sign/verify...)

Note: routing, timed messages, named client/server, scheduling models make a complex system (see the "big picture" in doc). The `oscbuildparse` module of `osc4py3` package can be used as is and provides nice OSC packets encoding/decoding functions usable in your own message transmission scheme.

The documentation is splitted in two main parts, a user guide for simple package use, and a developer guide explaining how things are organized and work together.

Contents

CHAPTER 1

Usage

1.1 Introduction

1.1.1 What is osc4py3?

It is a fresh implementation of the Open Sound Control (OSC) protocol[1] for Python3. It tries to have a *"pythonic"* programming interface at top layers, and to support different scheduling schemes in lower layers to have efficient processing.

1.1.2 What Python?

The package targets Python3.2 or greater, see other OSC implementations for older Python versions (OSC.py, simpleosc, TxOSC).

1.1.3 Why?

It was initially developed for use in a larger software as a background system communicating via OSC in separate threads, spending less possible time in main thread: sending OSC is just building messages/bundles objects and identifying targets, receiving OSC is just calling identified functions with OSC messages. All other operations, encoding and decoding, writing and reading, monitoring for communication channels availability, pattern matching messages addresses, etc, can be realized in other threads. On a multi-core CPU with multi-threading, these optimizations allow C code to run in parallel with communication operations, we gain milliseconds which are welcome in the project context.

1.1.4 What's new?

The whole package is really bigger and the underlying system is complex to understand. But, two base modules, `oscbuildparse` and `oscmethod` (see the modules documentation), are autonomous and can be used as is to

[1] See http://opensoundcontrol.org/ for complete OSC documentation.

provide: encoding and decoding of OSC messages and bundles, pattern matching and functions calling.

As we start from scratch, we use latest information about OSC with support for advanced data types (adding new types is easy), and insert support for OSC extensions like address pattern compression or messages control.

1.1.5 Complicated to use?

No. See the "Simple use" chapter below. It may be even easier to use than other Python OSC implementations. Don't mistake, the underlying processing is complex, but the high level API hide it.

1.2 Quick OSC

Open Sound Control is a simple way to do remote procedure calls between applications, triggered on string pattern matching, transported by any way inside binary packets. It mainly defines packets encoding format to transmit data and pattern matching to select functions to call. Using network broadcast or multicast, it allows to send data in one shot to several systems at same time.

1.2.1 Messages

Messages structure only contains: an **address string** (to be matched by patterns for routing to processing functions), **description of data** content (using type codes as defined in *Supported atomic data types*), and **data** itself (as packed binary).

For basic OSC address pattern, see *Address patterns*.

Data in OSC messages are generally simple, like can be function parameters, some int or float numbers, string, boolean, an "impulse" or "bang" (OSC come from MIDI (Musical Instrument Digital Interface)[2] musical world where a kind of "start" event is necessary). It could be used to transmit complex structures, but it's not its main purpose. See *Supported atomic data types* to see types allowed by `osc4py3`.

1.3 Simple use

1.3.1 Examples

We will first start by small commented examples, then give some explanations and precisions.

Note: Even if examples have been splitted in two parts, sending and receiving OSC messages can be realized in same program ; `osc_startup()`, `osc_process()` and `osc_terminate()` are client & server ready.

Receiving OSC messages

```
# Import needed modules from osc4py3
from osc4py3.as_eventloop import *
from osc4py3 import oscmethod as osm

def handlerfunction(s, x, y):
```

(continues on next page)

[2] See https://en.wikipedia.org/wiki/MIDI

(continued from previous page)

```
    # Will receive message data unpacked in s, x, y
    pass

def handlerfunction2(address, s, x, y):
    # Will receive message address, and message data flattened in s, x, y
    pass

# Start the system.
osc_startup()

# Make server channels to receive packets.
osc_udp_server("192.168.0.0", 3721, "aservername")
osc_udp_server("0.0.0.0", 3724, "anotherserver")

# Associate Python functions with message address patterns, using default
# argument scheme OSCARG_DATAUNPACK.
osc_method("/test/*", handlerfunction)
# Too, but request the message address pattern before in argscheme
osc_method("/test/*", handlerfunction2, argscheme=osm.OSCARG_ADDRESS + osm.OSCARG_
↪DATAUNPACK)

# Periodically call osc4py3 processing method in your event loop.
finished = False
while not finished:
    # ...
    osc_process()
    # ...

# Properly close the system.
osc_terminate()
```

Sending OSC messages

```
# Import needed modules from osc4py3
from osc4py3.as_eventloop import *
from osc4py3 import oscbuildparse

# Start the system.
osc_startup()

# Make client channels to send packets.
osc_udp_client("192.168.0.4", 2781, "aclientname")

# Build a simple message and send it.
msg = oscbuildparse.OSCMessage("/test/me", ",sif", ["text", 672, 8.871])
osc_send(msg, "aclientname")

# Build a message with autodetection of data types, and send it.
msg = oscbuildparse.OSCMessage("/test/me", None, ["text", 672, 8.871])
osc_send(msg, "aclientname")

# Buils a complete bundle, and postpone its executions by 10 sec.
exectime = time.time() + 10   # execute in 10 seconds
msg1 = oscbuildparse.OSCMessage("/sound/levels", None, [1, 5, 3])
msg2 = oscbuildparse.OSCMessage("/sound/bits", None, [32])
```

(continues on next page)

(continued from previous page)

```
msg3 = oscbuildparse.OSCMessage("/sound/freq", None, [42000])
bun = oscbuildparse.OSCBundle(oscbuildparse.unixtime2timetag(exectime),
                      [msg1, msg2, msg3])
osc_send(bun, "aclientname")

# Periodically call osc4py3 processing method in your event loop.
finished = False
while not finished:
    # You can send OSC messages from your event loop too...
    # ...
    osc_process()
    # ...

# Properly close the system.
osc_terminate()
```

You can take a look at the `rundemo.py` script in `demos/` directory. It is a common demo for the different scheduling options, where main `osc4py3` functions are highlighted by surrounding comments. You can also look at `speedudpsrv.py` and `speedudpcli.py` in the `demos/` directory.

General layout

So, the general layout for using `osc4py3` is

1. Import `osc4py3.as_xxx` module upon desired *threading model*.
2. Import *`osc4py3.oscbuildparse`* if you plan to send messages.
3. Call *`osc_startup`* function to start the system.
4. Create a set of client/server identified via channels names.
5. Register some handler functions with address matching.
6. Enter the processing loop.
   (a) Send created messages via client channels.
   (b) Call *`osc_process`* function to run the system.
   (c) Received messages from server channels, have their address matching handlers called for you.
7. Call *`osc_terminate`* to properly release resources (communications, threads...).

1.3.2 Sending messages

As seen in *Examples*, you need to import `oscbuildparse` module (or directly some of its content) to build a new *`OSCMessage`* object. Then, simply and send it via `osc_send()` specifying target client channel names (see *Client channels*).

```
msg = oscbuildparse.OSCMessage("/test/me", ",sif", ["text", 672, 8.871])
osc_send(msg, "aclientname")
```

You can see supported message data types in *Supported atomic data types*.

It is also possible to send multiple OSC messages, or to request an immediate or on the contrary differed execution time using *`OSCBundle`* (see *OSC Bundles*).

```
bun = oscbuildparse.OSCBundle(unixtime2timetag(time.time()+30),
        [oscbuildparse.OSCMessage("/test/me", ",sif", ["none", -45, 11.34]),
         oscbuildparse.OSCMessage("/test/him", ",sif", ["two", 15, 11.856])])
osc_send(bun, "aclientname")
bun = oscbuildparse.OSCBundle(.oscbuildparse.OSC_IMMEDIATELY,
        [oscbuildparse.OSCMessage("/test/me", ",sif", ["three", 92, 454]),
         oscbuildparse.OSCMessage("/test/him", ",sif", ["four", 107, -3e2])])
osc_send(bun, "aclientname")
```

Warning: Note that when running the system with `osc4py3.as_eventloop`, several calls to `osc_process()` may be necessary to achieve complete processing of message sending (the message itself is generally sent / received in one `osc_process()` call if possible, but some monitoring status on sockets may need other calls to be stopped, and several sent messages may take time to be processed on sockets).

1.3.3 Message handlers

A message handler is a Python callable (function or bound method, or Python partial), which will be called when an `OSCMessage`'s `addresspattern` is matched by a specified addrpattern filter. By default it is called with the (unpacked) OSC message data as parameters.

You can build message handlers with an interface specifying each independent argument, like:

```
def handler_set_location(x, y, z):
    # Wait for x,y,z of new location.
```

Or use of `*args` variable arguments count.

```
def handler_set_location(*args):
    # Wait for x,y,z of new location.
```

Hint: If you want your method to be called even if sender gives an invalid parameters count, you may prefer the variable arguments count solution or the `OSCARG_DATA` argscheme see below), else with an invalid argument count message, your handler call will fail immediately with a `ValueError` exception.

Installing message handlers

The normal way is to use `osc_method()` function, where you associate your handler function with an OSC address pattern filter string expression (see *Address patterns*).

```
def osc_method(addrpattern, function [,argscheme[,extra]])
```

There is by default a simple mapping of the OSC message data to your function parameters, but you can change this using the `argscheme` parameter (see *Arguments schemes*).

You can provide an extra parameter(s) for your handler function, given as `extra` (you must require this parameter in your `argscheme` too).

Address patterns

When calling *osc_method* function you specify the message address pattern filter as first parameter.

```
def osc_method(addrpattern, function [,argscheme[,extra=None]])
```

By default the `addrpattern` follows OSC pattern model[3] , which looks like unix/dos globbing for filenames:

- use / separators between "levels"
- * match anything, operator stop at / boundary
- // at beginning allow any level deep from start
- ? match any single char
- `[abcdef]` match any any char in abcdef
- `[a-z]` match any char from a to z
- `[!a-z]` match any char non from a to z
- `{truc,machin,chose}` match truc or machin or chose

Note: There is no unbind function at this level of `osc4py3` use.

Arguments schemes

By default message data are unpacked as handler parameters.

You can change what and how parameters are passed to handler's arguments with the `argscheme` parameter of *`osc_method`* function. This parameter uses some tuple constant from `oscmethod` module, which can be added to build your handler function required arguments list.

- `OSCARG_DATA` — OSC message data tuple as one parameter.
- `OSCARG_DATAUNPACK` — OSC message data tuple as N parameters (default).
- `OSCARG_MESSAGE` — OSC message as one parameter.
- `OSCARG_MESSAGEUNPACK` — OSC message as three parameters (addrpattern, typetags, arguments).
- `OSCARG_ADDRESS` — OSC message address as one parameter.
- `OSCARG_TYPETAGS` — OSC message typetags as one parameter.
- `OSCARG_EXTRA` — Extra parameter you provide as one parameter - may be None.
- `OSCARG_EXTRAUNPACK` — Extra parameter you provide (unpackable) as N parameters.
- `OSCARG_METHODFILTER` — Method filter object as one parameter. This argument is a `MethodFilter` internal structure containing informations about the message filter and your handler (argument scheme, filter expression...).
- `OSCARG_PACKOPT` — Packet options object as one parameter - may be None. This argument is a `PacketOption` internal structure containing information about packet processing (source, time, etc).
- `OSCARG_READERNAME` — Name of transport channel which receive the OSC packet - may be None.
- `OSCARG_SRCIDENT` — Indentification information about packet source (ex. (ip,port)) - may be None.
- `OSCARG_READTIME` — Time when the packet was read - may be None.

By combining these constants with **addition operator**, you can change **what arguments** your handler function will receive, and **in what order**. You can then adapt your handler function and/or the arguments scheme to fit. Examples:

[3] It is internally rewritten as Python `re` pattern to directly use Python regular expressions engine for matching.

```
# Import one of osc4py3 top level functions, and argument schemes definitions
from osc4py3.as_allthreads import *      # does osc_method
from osc4py3.oscmethod import *          # does OSCARG_XXX

# Request the unpacked data (default).
def fct(arg0, arg1, arg2, argn): pass
def fct(*args): pass
osc_method("/test/*", fct)

# Request the message address pattern and the unpacked data.
def fct(address, arg0, arg1, arg2, argn): pass
def fct(address, *args): pass
osc_method("/test/*", fct, argscheme=OSCARG_ADDRESS + OSCARG_DATAUNPACK)

# Request the message address pattern and the packed data.
def fct(address, args): pass
osc_method("/test/*", fct, argscheme=OSCARG_ADDRESS + OSCARG_DATA)

# Request the message address pattern, its type tag and the unppacked data.
def fct(address, typetag, arg0, arg1, arg2, argn): pass
def fct(address, typetag, *args): pass
osc_method("/test/*", fct, argscheme=OSCARG_ADDRESS + OSCARG_TYPETAGS + OSCARG_
↪DATAUNPACK)

# Request the message address pattern, its type tag and the packed data... the␣
↪OSCMessage parts.
def fct(address, typetag, args): pass
osc_method("/test/*", fct, argscheme=OSCARG_MESSAGEUNPACK)

# Request the OSCMessage as a whole.
def fct(msg): pass
osc_method("/test/*", fct, argscheme=OSCARG_MESSAGE)

# Request the message source identification, the whole message, and some extra data.
def fct(srcident, msg, extra): pass
osc_method("/test/*", fct, argscheme=OSCARG_SRCIDENT + OSCARG_MESSAGE + OSCARG_EXTRA,
           extra=dict(myoption1=4, myoption2="good"))
```

Hint: Informations made available via argument scheme constants should be enough for large common usage. But, if some handler function need even more information from received messages, other members of internal objects coming with `OSCARG_METHODFILTER` and `OSCARG_PACKOPT` `argscheme` can be of interest.

1.3.4 Threading model

From `osc4py3` package, you must import functions from one of the `as_eventloop`, `as_comthreads` or `as_allthreads` modules (line 2 in examples). Each of these modules publish the same set of `osc_...` functions for different threading models (star import is safe with `as_xxx` modules, importing only the `osc_fctxxx` functions).

All threads

Using `as_allthreads` module, threads are used in several places, including message handlers calls - by default a pool of ten working threads are allocated for message handlers calls.

To control the final execution of methods, you can add an `execthreadscount` named parameter to `osc_startup()`. Setting it to 0 disable executions of methods in working threads, and all methods are called in the context of the raw packets processing thread, or if necessary in the context of the delayed bundle processing thread. Setting it to 1 build only one working thread to process methods calls, which are then all called in sequence in the context of that thread.

If you want to protect your code against race conditions between your main thread and message handlers calls, you should better use `as_comthreads`.

Communication threads

Using `as_comthreads` module, threads are used for communication operations (sending and receiving, intermediate processing), and messages are stored in queues between threads.

Message handler methods are called only when you call yourself `osc_process()`. This is optimal if you want to achieve maximum background processing of messages transmissions, but final messages processing must occur within an event loop.

No thread

Using `as_eventloop` module, there is no multi-threading, no background processing. All operations (communications and message handling) take place only when you call `osc_process()`.

> **Warning:** Multi-threading add some overhead in the whole processing (due to synchronization between threads). By example, on my computer, transmitting 1000 messages each executing a simple method call, `speedudpsrv.py` goes from 0.2 sec with eventloop scheduling to 0.245 sec with comthreads scheduling and 0.334 sec with allthreads scheduling.
>
> But multi-threading can remain interesting if your main thread does C computing while osc4py3 process messages in background (which is the first use case of this development).

Important: If you choose to run the system with `as_eventloop` or with `as_comthreads`, you **must** periodically call the function `osc_process()` sometime in your event loop.

This call all code needed in the context of the event loop (all transmissions and dispatching code for `as_eventloop`, and only dispatching for `as_comthreads`).

If you choose to run the system with `as_allthreads`, you don't need to call `osc_process()` (this function is defined in this module too, but does nothing). But you should have checked that your code will not have problems with concurrent access to resources (if needed there is a way to simply install a message handler to be called with an automatic Lock).

1.3.5 Servers channels

Server channel are identified by names, but you should not have to use these names once the server channel has been created (unless you do advanced internal processing on incoming OSC packets).

In the example server channels are simple UDP ports which will be listened to read some incoming datagram, first server listen for data from a specific IPV4 network, second server listen on all IPV4 networks.

You can also create servers via `osc_broadcast_server()` or `osc_multicast_server()`.

1.3.6 Client channels

Client channels are identified by names, allowing *light coupling* in your code when sending OSC messages.

In the example client channels is a simple UDP transport. It will transmit data to server channels listening on the other side.

You can also create clients via `osc_broadcast_client()` or `osc_multicast_client()` (providing ad-hoc broadcast/multicast addresses).

1.3.7 Light coupling

Client and server channels being identified via **names**, where name resolution is only done when communication activity is required, creation order of the different parts is not important.

In this way, in your code you send a message to a named channel, without knowing how and where this channel connection has been established. You can change transport protocol with no impact on your communication code. If you target multiple channels via a list or tuple or set, eventually nested — this allow to easily define groups of channels as target for sending.

If you use the special reserved `"_local"` name target, you send a message to be processed directly by your own program without network communication (no need to create a channel, it's a bypass name).

1.3.8 Logging OSC operations

Once imported, use of the package begin by initializing the system with a call to `osc_startup()` function.

This function allows an optional `logger` named parameter, accepting a `logging.logger` object[4] . Example:

```
import logging
logging.basicConfig(format='%(asctime)s - %(threadName)s ø %(name)s - '
    '%(levelname)s - %(message)s')
logger = logging.getLogger("osc")
logger.setLevel(logging.DEBUG)
osc_startup(logger=logger)
```

1.3.9 Advanced pattern/handler

You may setup your pattern / handler binding using low level functions, giving access to some more advanced options.

```
from osc4py3 import oscmethod, oscdispatching
mf = oscmethod.MethodFilter(...)
oscdispatching.register_method(mf)
```

By creating the `osc4py3.oscmethod.MethodFilter` object directly, you can provide some extra construction parameters which are not available using high level `osc_method()` function (we don't list parameters which can be specified using `argscheme`):

- `patternkind` default to `"osc"`, indicates to use OSC syntax for pattern. You can use here `"re"` to specify that you provide in `addrpattern` a Python regular expression.
- `workqueue` allows you to specify a processing `WorkQueue` associated to threads to call the handler with matched incoming messages — by default it's `None` and indicate to use the common work queue.

[4] This is highly recommended in case of problems with multi-threads usage.

- `lock` can be used to specify a `thread.Lock` (or `thread.RLock`) object to acquire before processing matched messages with this handler function — by default it's `None`.
- `logger` is a specific logger to use to specifically trace that handler calling on matched messages reception. By default top level `osc_method()` functions use the logger given as `osc_startup() logger` parameter.

1.4 User main functions

See `OSCMessage` or `OSCBundle` for their respective documentation.

We document using the common `as_xxx` module

Note: Startup and termination functions (`osc_startup()` and `osc_terminate()`) are intended to be called once only, at begining of program and at end of program. They install/uninstall a set of communication and filtering tools which normally stay active during program execution.

Terminating then restarting osc4py3 may work... but has not been extensively tested.

Each `as_eventloop`, `as_allthreads` and `as_comthreads` module define the same set of functions documented here.

`osc4py3.as__common.`**`osc_startup`**(***kwargs*)

Once call startup function for all osc processing in the event loop.

Create the global dispatcher and register it for all packets and messages. Create threads for background processing when used with as_allthreads or as_comthreads scheduling.

Parameters

- **`logger`** (*`logging.Logger`*) – Python logger to trace activity. Default to None
- **`execthreadscount`** (*`int`*) – number of execution threads for methods to create. Only used with as_allthreads scheduling. Default to 10.
- **`writethreadscount`** (*`int`*) – number of write threads for packet sending to create. Only used with as_allthreads scheduling. Default to 10.

`osc4py3.as__common.`**`osc_terminate`**()

Once call termination function to clean internal structures before exiting process.

`osc4py3.as__common.`**`osc_process`**()

Function to call from your event loop to receive/process OSC messages.

`osc4py3.as__common.`**`osc_method`**(*addrpattern*, *function*, *argscheme='data_unpack'*, *)*, *extra=None*)

Add a method filter handler to automatically call a function.

Note: There is no unregister function at this level of osc4py use, but module `oscdispatching` provides `MethodFilter` object and functions to register and unregister them.

Parameters

- **`addrpattern`** (*`str`*) – OSC pattern to match
- **`function`** (*`callable`*) – code to call with the message arguments

- **argscheme** (*tuple*) – scheme for handler function arguments. By default message data are transmitted, flattened as N parameters.
- **extra** (*anything*) – extra parameters for the function (must be specified in argscheme too).

osc4py3.as__common.**osc_send**(*packet*, *names*)
Send the packet using channels via names.

Parameters

- **packet** (*OSCMessage or OSCBundle*) – the message or bundle to send.
- **names** (*str or list or set*) – name of target channels (can be a string, list or set).

osc4py3.as__common.**osc_udp_server**(*name*, *address*, *port*)
Create an UDP server channel to receive OSC packets.

Parameters

- **name** (*str*) – internal identification of the channel server.
- **address** (*str*) – network address for binding UDP socket
- **port** (*int*) – port number for binding UDP port

osc4py3.as__common.**osc_udp_client**(*name*, *address*, *port*)
Create an UDP client channel to send OSC packets.

Parameters

- **name** (*str*) – internal identification of the channel client.
- **address** (*str*) – network address for binding UDP socket
- **port** (*int*) – port number for binding UDP port

osc4py3.as__common.**osc_multicast_server**(*name*, *address*, *port*)
Create a multicast server to receive OSC packets.

Parameters

- **name** (*str*) – internal identification of the channel server.
- **address** (*str*) – network address for binding socket
- **port** (*int*) – port number for binding port

osc4py3.as__common.**osc_multicast_client**(*name*, *address*, *port*, *ttl*)
Create a multicast client channel to send OSC packets.

Parameters

- **name** (*str*) – internal identification of the channel client.
- **address** (*str*) – multicast network address for binding socket
- **port** (*int*) – port number for binding port
- **ttl** (*int*) – time to leave for multicast packets. Default to 1 (one hop max).

osc4py3.as__common.**osc_broadcast_server**(*name*, *address*, *port*)
Create a broadcast server channel to receive OSC packets.

Parameters

- **name** (*str*) – internal identification of the UDP server.

- **address** (*str*) – network address for binding UDP socket
- **port** (*int*) – port number for binding UDP port

osc4py3.as__common.**osc_broadcast_client**(*name*, *address*, *port*, *ttl*)
Create a broadcast client channel to send OSC packets.

Parameters

- **name** (*str*) – internal identification of the channel client.
- **address** (*str*) – broadcast network address for binding socket
- **port** (*int*) – port number for binding port
- **ttl** (*int*) – time to leave for broadcast packets. Default to 1 (one hop max).

CHAPTER 2

Messages and Bundles

Note: OSC structures are defined in module `oscbuildparse`. This module provides all low level tools to manipulate these structures, build them, encode them to raw OSC packets, and decode them back.

You may specially be interested by message construction in the examples (encoding and decoding is normally done automatically for you by `osc4py3`).

This module is only here to translate OSC packets from/to Python values. It can be reused anywhere as is (only depend on Python3 standard modules).

Examples:

```
>>> from osc4py3.oscbuildparse import *
>>> dir()
['OSCBundle', 'OSCCorruptedRawError', 'OSCError', 'OSCInternalBugError',
'OSCInvalidDataError', 'OSCInvalidRawError', 'OSCInvalidSignatureError',
'OSCMessage', 'OSCUnknownTypetagError', 'OSC_BANG', 'OSC_IMMEDIATELY',
'OSC_IMPULSE', 'OSC_INFINITUM', 'OSCbang', 'OSCmidi', 'OSCrgba', 'OSCtimetag',
'__builtins__', '__doc__', '__name__', '__package__', 'decode_packet',
'dumphex_buffer', 'encode_packet', 'float2timetag', 'timetag2float',
'timetag2unixtime', 'unixtime2timetag']

>>> msg = OSCMessage('/my/pattern',',iisf',[1,3,"a string",11.3])
>>> raw = encode_packet(msg)
>>> dumphex_buffer(raw)
000:2f6d792f 70617474 65726e00 2c696973     /my/ patt ern. ,iis
016:66000000 00000001 00000003 61207374     f... .... .... a st
032:72696e67 00000000 4134cccd              ring .... A4..
>>> decode_packet(raw)
OSCMessage(addrpattern='/my/pattern', typetags=',iisf', arguments=(1, 3,
'a string', 11.300000190734863))

>>> import time
>>> bun = OSCBundle(unixtime2timetag(time.time()+1),
```

(continues on next page)

(continued from previous page)

```
        [OSCMessage("/first/message",",ii",[1,2]),
         OSCMessage("/second/message",",fT",[4.5,True])])
>>> raw = encode_packet(bun)
>>> dumphex_buffer(raw)
000:2362756e 646c6500 d2c3e04f 455a9000     #bun dle. ...O EZ..
016:0000001c 2f666972 73742f6d 65737361     .... /fir st/m essa
032:67650000 2c696900 00000001 00000002     ge.. ,ii. .... ....
048:00000018 2f736563 6f6e642f 6d657373     .... /sec ond/ mess
064:61676500 2c665400 40900000              age. ,fT. @...
>>> decode_packet(raw)
OSCBundle(timetag=OSCtimetag(sec=3536052435, frac=3018637312),
elements=(OSCMessage(addrpattern='/first/message', typetags=',ii',
arguments=(1, 2)), OSCMessage(addrpattern='/second/message', typetags=',fT',
arguments=(4.5, True))))

>>> msg = OSCMessage("/shortcut/with/typedetection", None,
        [True, OSC_BANG, 12, 11.3])
>>> raw = encode_packet(msg)
>>> dumphex_buffer(raw)
000:2f73686f 72746375 742f7769 74682f74     /sho rtcu t/wi th/t
016:79706564 65746563 74696f6e 00000000     yped etec tion ....
032:2c544969 66000000 0000000c 4134cccd     ,TIi f... .... A4..
>>> decode_packet(raw)
OSCMessage(addrpattern='/shortcut/with/typedetection', typetags=',TIif',
arguments=(True, OSCbang(), 12, 11.300000190734863))
```

Note : you can find other examples in `osc4py3/tests/buildparse.py` module.

2.1 Programming interface

There are two main functions for advanced users:

- *`encode_packet()`* build the binary representation for OSC data
- *`decode_packet()`* retrieve OSC data from binary representation

An OSC packet can either be an OSC message, or an OSC bundle (which contains a collection of messages and bundles - recursively if needed).

2.1.1 OSC Messages

For developer point of view, there is an *`OSCMessage`* named tuple class which is used as container to encode and decode messages. It contains fields accordingly to OSC1.1 protocol:

class osc4py3.oscbuildparse.**OSCMessage**
`OSCMessage(addrpattern, typetags, arguments)` → named tuple

Variables

- **addrpattern** (*string*) – a string beginning by `/` and used by OSC dispatching protocol.
- **typetags** (*string*) – a string beginning by `,` and describing how to encode values
- **arguments** (*list* / *tuple*) – a list or tuple of values to encode.

The `typetag` must start by a comma (`','`) and use a set of chars to describe OSC defined data types, as listed in the *Supported atomic data types* table. It may optionally be set to `None` for an automatic detection of type tags from values (see *Automatic type tagging* for detection rules).

2.1.2 OSC Bundles

And to add a time tag or group several messages in a packet, there is an *`OSCBundle`* named tuple which is used to encode and decode bundles. It contains fields accordingly to OSC1.1 protocol:

class osc4py3.oscbuildparse.**OSCBundle**
`OSCBundle(timetag, elements)` → named tuple

Variables

- **`timetag`** – a time representation using two int values, sec:frac
- **`elements`** (*`list`*|*`tuple`*) – a list or tuple of mixed OSCMessage / OSCBundle values

Its first `timetag` field must be set to `osc4py3.oscbuildparse.OSC_IMMEDIATELY` to request an immediate processing of the bundle messages by the server's matching handlers. Else, it is considered as an OSC time (see *Time Tag*) and must be computed for planned processing time.

2.1.3 Supported atomic data types

In addition to the required OSC1.1 `ifsbtTFNI` type tag chars, we support optional types of OSC1.0 protocol `hdScrm[]` (support for new types is easy to add if necessary).

Attention: Check that programs receiving your data also support optional data types. You may use the *Out of band options restrict_typetags* to limit the data types manipulated by `osc4py3`.

Table 1: Type codes

Tag	Data	Python	Notes
i	int32	`int`	signed *integer*
f	float32	`float`	a C float (4 bytes)
s	string	`str`	ASCII *string*
b	blob	`memoryview`	mapping to part in received *blob* data
h	int64	`int`	to transmit larger integers
t	timetag	*`OSCtimetag`*	two bytes named tuple time
d	float64	`float`	a C double (8 bytes)
S	alt-string	`str`	ASCII strings to distinguish with 's' *strings*
c	ascii-char	`str`	one ASCII *char*
r	rgba-color	`OSCrgba`	four bytes fields named tuple *RGBA data*
m	midi-msg	`OSCmidi`	four bytes fields named tuple *MIDI data*
T	(none)	`True`	direct True value only with type tag
F	(none)	`False`	direct False value only with type tag
N	(none)	`None`	'nil' in OSC only with type tag
I	(none)	`OSCbang`	named tuple with no field (see *bang*)
[	(none)	`tuple`	beginning of an *array*
]	(none)		end of the *array* (to terminate tuple)

Integer

Care that Python int has a range with "no limit". Overflows will only be detected when trying to pack such values into the 32 bits representation of an OSC packet integer.

String and char

They are normally transmitted as ASCII, default processing ensure this encoding (with `strict` error handling, raising an exception if a non-ASCII char is in a string). An `oob` option (see *oob options* below) allows to specify an encoding.

The value for a string/char is normally a Python `str` ; you can give a `bytes` or `bytearray` or `memoryview`, but they must not contain a zero byte (except at the end - and it will be used a string termination when decoding).

For a char, the `string` / `bytes` / ... must contain only one element.

In OSC, distinction between `s` strings and `S` strings are application meaning defined. And in `osc4py3` you don't have direct access to data type codes (you may request to get the .

Blob

They allow transmission of any binary data of your own. The value for a blob can be `bytes` or `bytearray` or `memoryview`.

When getting blob values on packet reception, they are returned as `memoryview` objects to avoid extra data copying. You may cast them to `bytes` if you want to copy/extract the value from the OSC packet.

Infinitum / Impulse / Bang

The Infinitum data (`'I'`), renamed as Impulse 'bang' in OSC 1.1, is returned in Python as a `OSCbang` named tuple (with no value in the tuple). Constant `OSC_INFINITUM` is defined as an `OSCbang` value, and aliases constants `OSC_IMPULSE` and `OSC_BANG` are also defined.

You should test on object class with `isinstance(x,OSCbang)`.

Time Tag

As time tag is stored into an *OSCtimetag* named tuple with two items.

class osc4py3.oscbuildparse.**OSCtimetag**

OSCtimetag(sec, frac) → named tuple

Time tags are represented by a 64 bit fixed point number of seconds relative to 1/1/1900, same as Internet NTP timestamps .

> **Warning:** We don't check that sec and frac parts fill in 32 bits integers, this is detected by struct.pack() function.

Attribute int sec first 32 bits specify the number of seconds since midnight on January 1, 1900,

Attribute int frac last 32 bits specify fractional parts of a second to a precision of about 200 picoseconds.

As it is not really usable with usual Python time, four conversion functions have been defined:

- *timetag2float()* convert an *OSCtimetag* tuple into a float value in seconds from 1/1/1900,
- *timetag2unixtime()* convert an *OSCtimetag* tuple into a Unix float time in seconds from 1/1/1970 (Python time),
- *float2timetag()* convert a float value of seconds from 1/1/1900 into an *OSCtimetag* tuple,
- *unixtime2timetag()* convert a Unix float value of seconds from 1/1/1970 (Python time) into an *OSCtimetag* tuple - can be used

The special value used in OSC to indicate an "immediate" time, with a time tag having 0 in seconds field and 1 in factional part field (represented as 0x00000001 value), is available for comparison and usage in constant `osc4py3.oscbuildparse.OSC_IMMEDIATELY`.

Array

An array is a way to group some data in the OSC message arguments. On the Python side an array is simply a list or a tuple of values. By example, to create a message with two int followed by four grouped int, you will have:

- Type tags string: `',ii[iiii]'`
- Arguments list: `[3, 1, [4, 2, 8, 9]]`

Note : When decoding a message, array arguments are returned as tuple, not list. In this example: `(3, 1, (4, 2, 8, 9))`.

RGBA data

RGBA values are stored into an `OSCrgba` named tuple containing four single byte values (int in 0..255 range):

0. `red`
1. `green`
2. `blue`
3. `alpha`

MIDI data

MIDI values are stored into an `OSCmidi` named tuple containing four single byte values (int in 0..255 range):

0. `portid`
1. `status`
2. `data1`
3. `data2`

OSCbang = namedtuple('OSCbang', '')

2.1.4 Automatic type tagging

When creating an *OSCMessage*, you can give a `None` value as typetags. Then, message arguments are automatically parsed to identify their types and build the type tags string for you.

The following mapping is used:

Table 2: Automatic typing

What	Type tag and corresponding data
value `None`	`N` without data
value `True`	`T` without data
value `False`	`F` without data
type `int`	`i` with int32
type `float`	`f` with float32
type `str`	`s` with string
type `bytes`	`b` with raw binary
type `bytearray`	`b` with raw binary
type `memoryview`	`b` with raw binary
type `OSCrgba`	`r` with four byte values
type `OSCmidi`	`m` with four byte values
type `OSCbang`	`I` without data
type *`OSCtimetag`*	`t` with two int32

2.2 Errors

All errors explicitly raised by the module use specify hierarchy of exceptions:

```
Exception
  OSCError
    OSCCorruptedRawError
    OSCInternalBugError
    OSCInvalidDataError
    OSCInvalidRawError
    OSCInvalidSignatureError
    OSCUnknownTypetagError
```

2.2.1 OSCError

This is the parent class for OSC errors, usable as a catchall for all errors related to this module.

2.2.2 OSCInvalidDataError

There is a problem in some OSC data provided for encoding to raw OSC representation.

2.2.3 OSCInvalidRawError

There is a problem in a raw OSC buffer when decoding it.

2.2.4 OSCInternalBugError

Hey, we detected a bug in OSC module. Please, signal it with description of the context, data processed, options used.

2.2.5 OSCUnknownTypetagError

Found an invalid (unknown) type tag when encoding or decoding. This include type tags not in a subset with *restrict_typetags* option.

2.2.6 OSCInvalidSignatureError

Check of raw data with signature failed due bad source or modified data. This can only occur with advanced packet control enabled and signature functions installed in out-of-band.

2.2.7 OSCCorruptedRawError

Check of raw data with checksum failed. This can only occur with advanced packet control enabled and checksum functions installed in out-of-band.

2.3 Out of band options

Warning: Out of band options may need further debugging.

These OOB options are transmitted as a simple Python dict among internal `oscbuildparse` functions to enable and define parameters of extra processing.

You may add your own keys in this dict to transmit data to your extra processing functions.

2.3.1 Supported data types

restrict_typetags

`oob['restrict_typetags']` must contain a string with the subset typecode chars you want to allow (in `ifsbtTFNIhdScrm[]`) — see *Supported atomic data types*. This make your program ensure that it don't use data types unsupported by other OSC implementations.

2.3.2 Strings encoding

The OSC standard encode strings as ASCII (American Standard Code for Information Interchange) only chars, which include control chars (codes 1 to 31 and 127; code 0 is used as end of string marker), whitespace and following printable chars: !"#$%&'()*+,-./ 0123456789 :;<=>?@ ABCDEFGHIJKLMNOPQRSTUVWXYZ []^_‘ abcdefghijklmnopqrstuvwxyz {|}~

This is how `osc4py3` works, converting all Unicode Python strings into an ASCII encoding. By default the encoding and decoding use `strict` error handling scheme: any out of ASCII char in a Python string raise an `UnicodeEncodeError` when encoding to OSC sctring, and any non-ASCII char in an OSC string coming from somewhere raise an `UnicodeDecodeError` when decoding to Python string.

You can modify the encoding to use and the error processing scheme (`strict`, `replace`, `ignore`...) with following OOB options.

Caution: Changing strings encoding to non-ASCII goes out of OSC standards, you may brake communications with other OSC implementations. It may be better to transmit encoded text in blobs and to agree on encoding on both sides (ex. transmit encoding in a separate OSC string).

str_decode

`oob['str_decode']` must contain a tuple of two values to specify how to decode OSC strings. First item is the encoding to use, second item the error handling scheme. Default to `('ascii', 'strict')`.

str_encode

`oob['str_encode']` must contain a tuple of two values to specify how to encode OSC strings. First item is the encoding to use, second item the error handling scheme. Default to `('ascii', 'strict')`.

char_decode

`oob['char_decode']` must contain a tuple of two values to specify how to decode OSC char. First item is the encoding to use, second item the error handling scheme. Default to `('ascii', 'strict')`.

char_encode

`oob['char_encode']` must contain a tuple of two values to specify how to encode OSC char. First item is the encoding to use, second item the error handling scheme. Default to `('ascii', 'strict')`.

2.3.3 Compression of addresses

Note: The `osc4py3` package implement support for OSC address compression as presented in Improving the Efficiency of Open Sound Control (OSC) with Compressed Address Strings (SMC 2011, by Jari Kleimola and Patrick J. McGlynn).

Two sides of an OSC communication agree on some int to string mapping. The address is then sent as a single `"/"` OSC string followed by a 32 bits int code.

This implementation only do compression / decompression of addresses, it's up to the user to exchange int / address mapping — by example via an initial exchange of OSC messages (eventually grouped in a bundle).

addrpattern_decompression

`oob['addrpattern_decompression']` must contain the int to string mapping to retrieve message address from int code for incoming packets.

addrpattern_compression

`oob['addrpattern_decompression']` must contain the string to int mapping to get int code from message address for outgoing packets.

2.3.4 Basic messages controls

check_addrpattern

This option allows to check that address string correctly follow OSC pattern.

`oob['check_addrpattern']` must be a boolean set to `True` to check that address string correctly follow OSC pattern.

force_typetags

This option force presence of type tags in received OSC packets, you can't send an only address message with this option enabled (it must contain at least a `","` string indicating no data).

`oob['force_typetags']` must be a boolean set to `True` to force presence of a type tags specification, even with no data (in such case type tags simply contains `","` string).

2.3.5 Dump of packets

decode_packet_dumpraw

`oob['decode_packet_dumpraw']` must be a boolean set to `True` to enable file writing of raw OSC packets in hexadecimal representation.

encode_packet_dumpraw

`oob['encode_packet_dumpraw']` must be a boolean set to `True` to enable file writing of raw OSC packets in hexadecimal representation.

decode_packet_dumpacket

`oob['decode_packet_dumpacket']` must be a boolean set to `True` to enable file writing of decoded OSC packets representation (OSCMessage or OSCBundle).

encode_packet_dumpacket

`oob['encode_packet_dumpacket']` must be a boolean set to `True` to enable file writing of encoded OSC packets representation (OSCMessage or OSCBundle).

dump_decoded_values

`oob['dump_decoded_values']` must be a boolean set to `True` to enable file writing of individual fields of decoded message data .

dumpfile

`oob['dumpfile']` must be a writable stream (file...), used to dump OSC packets (raw or decoded) when *decode_packet_dumpraw* or *decode_packet_dumpacket encode_packet_dumpraw* or *encode_packet_dumpacket* is enabled.

If it is not defined, packets are dump to `sys.stdout` stream.

2.3.6 Advanced control

Note: Following OOB options have been installed to setup controlled OSC communications, with possible encryption of data, authentication, checksum...

In such situation OSC messages have an address string set to `"/packet"`, and data is a set of 6 data corresponding to:

0. `cheksumprot` a string indicating checksum to use
1. `rawcksum` a blob with checksum
2. `authprot` a string indicating authentication to use
3. `rawckauth` a blob with authentication token
4. `cryptprot` a string indicating encryption to use
5. `rawoscdata` a blob containing (encrypted) data

Once the message have passed all steps, a normal OSC message is retrieved (which can go normally in `osc4py3` pipeline).

When receiving a packet, data is decoded then authentified then sumchecked.

When sending a packet, data is sumchecked then authentified then encoded.

If a support function is not present in the oob, its feature is simply ignored.

When advanced control functions receive raw data, it's a memoryview (on the rawoscdata blob).

advanced_packet_control

`oob['advanced_packet_control']` must be a boolean set to `True` to enable packets control.

packet_crypt_prot

`oob['packet_crypt_prot']` may contain string indicating encryption protocol to use. It default to empty string.

packet_encrypt_fct

`oob['packet_encrypt_fct']` is a function to encode raw osc data. It is called with the data to encode, indication of the encryption protocol, and the oob. It must return binary representation of encoded data. Call example:

```
tobuffer = fencrypt(tobuffer, cryptprot, oob)
```

packet_decrypt_fct

`oob['packet_decrypt_fct']` is a function to decode raw osc data. It is called with the data to decode, indication of the encryption protocol, and the oob. It must return binary representation of decoded data. Call example:

```
rawoscdata = fdecrypt(rawoscdata, cryptprot, oob)
```

packet_authsign_prot

`oob['packet_authsign_prot']` may contain string indicating authentication protocol to use. It default to empty string.

packet_mkauthsign_fct

`oob['packet_mkauthsign_fct']` is a function to build authentication data. It is called with osc data buffer and the authentication protocol. It must return the authentication value to transmit as a blob compatible data. Call example:

```
authsign = fauthsign(tobuffer, authprot, oob)
```

packet_ckauthsign_fct

`oob['packet_ckauthsign_fct']` is a function to check authentication. It is called with (decoded) osc data, the two authentication fields and the oob. It must simply raise an exception if authentication is not proven. Call example:

```
fckauthsign(rawoscdata, rawckauth, authprot, oob)
```

packet_checksum_prot

`oob['packet_checksum_prot']` may contain string indicating checksum protocol to use. It default to empty string.

packet_mkchecksum_fct

`oob['packet_mkchecksum_fct']` is a function to build data integrity checksum value. It is called with osc data buffer, the checksum protocol and the oob. It must return the checksum value to transmit as a blob compatible data. Call example:

```
cksum = fchecksum(tobuffer, cksumprot, oob)
```

packet_ckcheksum_fct

`oob['packet_ckauthsign_fct']` is a function to check data integrity. It is called with (decoded) osc data, the two checksum fields and the oob. It must simply raise an exception if checksum is not verified. Call example:

```
fchecksumcheck(rawoscdata, rawcksum, cheksumprot, oob)
```

2.4 Code documentation

osc4py3.oscbuildparse.**decode_packet**(*rawoscdata*, *oob=None*)
From a raw OSC packet, extract the list of OSCMessage.

Generally the packet come from an OSC channel reader (UDP, multicast, USB port, serial port, etc). It can contain bundle or message. The function guess the packet content and call ah-hoc decoding.

This function map a memoryview on top of the raw data. This allow sub-called functions to not duplicate data when processing. You can provide directly a memoryview if you have a packet from which just a part is the osc data.

Parameters

- **rawoscdata** (*bytes or bytearray or memoryview (indexable bytes)*) – content of packet data to decode.
- **oob** (*dict*) – out of band extra parameters (see *Out of band options*).

Returns decoded OSC messages from the packet, in decoding order.

Return type [*OSCMessage*]

osc4py3.oscbuildparse.**encode_packet**(*content*, *oob=None*)
From an OSCBundle or an OSCMessage, build OSC raw packet.

Parameters

- **content** (OSCMessage *or* OSCBundle) – data of packet to encode
- **oob** (*dict*) – out of band extra parameters (see *Out of band options*).

Returns raw representation of the packet

Return type bytearray

osc4py3.oscbuildparse.**dumphex_buffer**(*rawdata*, *tofile=None*)
Dump hexa codes of OSC stream, group by 4 bytes to identify parts.

Parameters

- **data** (*bytes*) – some raw data to format.
- **tofile** (*file (or file-like)*) – output stream to receive dump

osc4py3.oscbuildparse.**timetag2float**(*timetag*)
Convert a timetag tuple into a float value in seconds from 1/1/1900.

Parameters **timetag** (OSCtimetag) – the tuple time to convert

Returns same time in seconds, with decimal part

Return type float

osc4py3.oscbuildparse.**timetag2unixtime**(*timetag*)
Convert a timetag tuple into a float value of seconds from 1/1/1970.

Parameters **timetag** (OSCtimetag) – the tuple time to convert

Returns time in unix seconds, with decimal part

Return type float

osc4py3.oscbuildparse.**float2timetag**(*ftime*)
Convert a float value of seconds from 1/1/1900 into a timetag tuple.

Parameters **ftime** (*float*) – number of seconds to convert, with decimal part

Returns same time in sec,frac tuple

Return type *OSCtimetag*

osc4py3.oscbuildparse.**unixtime2timetag**(*ftime=None*)
Convert a float value of seconds from 1/1/1970 into a timetag tuple.

Parameters **ftime** (*float*) – number of seconds to convert, with decimal part. If not specified, the function use current Python time.time().

Returns same time in sec,frac tuple

Return type *OSCtimetag*

CHAPTER 3

Development

You may keep the `osc4py3-bigpicture` graphic on hand. It shows the general organization of the packages, classes, functions, how they interact, how data are transmitted between the different layers. The `osc4py3` package is cut among:

- two OSC core modules for packets manipulation and methods routing: `oscbuildparse` and `oscmethod` ;
- four modules to run processing: `oscchannel`, `oscscheduling`, `oscdistributing` and `oscdispatching` ;
  - Module `oscchannel` manage transport (emission, reception) of raw OSC packets. It uses monitoring tools from `oscscheduling` to get information of incoming data or connexion availability for outgoing data.
  - Module `oscdistributing` transform OSC message and bundles into/from OSC raw packets and provide them to ad-hoc objects/functions for processing.
  - Module `oscdispaching` identify methods to call from messages address pattern matching and control delayed processing of bundle with future time tag.
- helped by specialized tool modules, by transport protocol specific modules and by a set of user helper modules providing different scheduling policies.

3.1 Implementation Modules

The two modules described here may be used without core modules if you require a simpler implementation of communications and distribution of OSC packets.

3.1.1 oscbuildparse

This module has an extensive documentation string you are invited to read.

Out-Of-Band options

These options are transmitted among building and parsing functions to activate / deactivate some processing alternatives. Options (keys, type, usage) are listed in the module documentation.

That way, you can control encoding, trace data, restrict supported type tags, activate address pattern compression, enable checksum or authentication signature or encryption, etc (warning : some code is untested).

3.1.2 oscmethod

This module has an extensive documentation string you are invited to read.

Pattern matching can use two syntax, OSC defined syntax or Python regular expression syntax (former is rewritten as later).

You basically create a MethodFilter object with at least an address pattern (a string) and a callable object. By default address pattern use OSC messages patterns syntax (parameter patternkind)

3.2 Core Modules

3.2.1 oscchannel

Organize communication channels to send and receive OSC packets from different transport protocols. `TransportChannel` provides an abstract class with ad-hoc interfaces. It gives some common services to subclasses and allow organization of sockets management to avoid multiplying threads (use of select or ad-hoc platform specific socket monitoring service upon a scheduling scheme). Specific handlers can be installed at transport channel object level, to have special operations occurring at identified processing time of in/out data (see actions_handlers).

3.2.2 oscscheduling

This module does the real job of monitoring the transport channels and transmitting data

3.2.3 oscdistributing

3.2.4 oscdispatching

3.3 Specialized Tools

3.3.1 oscpacketoptions

3.3.2 osctoolspools

3.3.3 oscnettools

3.4 Transport Protocol

3.4.1 Base transport class

Each transport protocol has its own module defining one or more `TransportChannel` subclasses. All these subclasses use the same construction scheme:

```
channel = ChannelClass("channelname", mode, { 'option_key': optionvalue })
```

The mode is a combination of `'r'` and `'w'` for read and write channels (ie. OSC servers to receive/read packets and OSC clients to send/write packets), and `'e'` for stream based channels waiting for connection event.

Common channel options

Most parameters of channels are set via the third parameter in the options dictionary. Here is a description of possible keys and values (with default value) :

- `auto_start` (`True`) bool flag to immediately activate the channel and start monitoring its activity once is has been initialized (else you must call yourself activate() and begin_scheduling() methods)
- `logger` (`None`) Python logging.Logger to trace activity. If left to None, there is almost no overhead... but you silently pass all errors. You may better setup one logger with an `logging.ERROR` filtering level.
- `actions_handlers` ({}) map { str : callable / str } of action verb and code or message to call, see "Action Handlers" below. For advanced usage with personal hacks at some processing times.
- `monitor` (Monitoring) monitor used for this channel - the channel register itself among this monitor when becoming scheduled. This is the tool used to detect state modification on the channel and activate reading or writing of data.
- `monitor_terminate` (`False`) bool flag to indicate that our monitor must be terminated with the channel deletion. Default upon automatic monitor, or use monitor_terminate key in options.
- `read_forceident` (`None`) map { source : identification } informations to use for peer identification on read data (don't use other identification ways).
- `read_dnsident` (`True`) bool flag to use address to DNS mapping automatically built from `oscnettools` module.
- `read_datagram` (`False`) bool flag to consider received data as entire datagrams (no data remain in the buffer, its all processed when received).

- `read_maxsources` (500) int count of maximum different sources allowed to send data to this reader simultaneously, used to limit an eventuel DOS on the system by sending incomplete packets. Set it to 0 to disable the limit. Default to `MAX_SOURCES_ALLOWED` (500).
- `read_withslip` (`False`) bool flag to enable SLIP protocol on received data.
- `read_withheader` (`False`) bool flag to enable detection of packet size in the beginning of data, and use read_headerunpack.
- `read_headerunpack` (`None`) (str,int,int) for automatic use of `struct.unpack()` or `fct(data)` → `packsize,headsize` to call a function. For `struct.unpack()`, data is a tuple with: the format to decode header, the fixed header size, and the index of packet length value within the header tuple returned by unpack(). For function, it will receive the currently accumulated data and must return a tuple with: the packet size extracted from the header, and the total header size. If there is no enough data in header to extract packet size, the function must return (0, 0). Default to `("!I", 4, 0)` to detect a packet length encoded in 4 bytes unsigned int with network bytes order. If the function need to pass some data in the current buffer (ex. remaining of an old failed communication), it can return an header size corresponding to the bytes count to ignore, and a packet size of 0 ; this will consume data with an -ignored- empty packet.
- `read_headermaxdatasize` (1 MiB) int maximum count of bytes allowed in a packet size field when using headers. Set it to 0 to disable the limit. Default to `MAX_PACKET_SIZE_WITH_HEADER` (1 MiB).
- `write_workqueue` (`None`) WorkQueue queue of write jobs to execute. This allow to manage initial writing to peers in their own threads (nice for blocking write() calls) or in an event loop if working without thread. The queue will be filled when we detect that a write can occur (ie same channel will have maximum one write operation in the workqueue, even if there are multiple pending operations in the write_pending queue).
- `write_wqterminate` (`False`) bool flag to indicate to call work queue termination when terminating the channel.
- `write_withslip` (`False`) bool flag to enable SLIP protocol on sent data.
- `write_slip_flagatstart` (`True`) bool flag to insert the SLIP END code (192) at beginning of sent data (when using SLIP).

Network address options

For channels using network address and port, the `oscnettools` module provide a common way to retrieve these informations. This is done via a options with variable prefix :

- `<prefix>_host` (no default) str representing an host, as a host name resolved via DNS, or as an IP address in IPV4 format or IPV6 format. Can use `"*"` string to specify wildcard address (ex. use with TCP to have server socket on all networks).
- `<prefix>_port` (no default) int or str representing a network port number or service name. Can use 0 integer or `"None"` string to specify random port.
- `<prefix>_forceipv4` (`False`) bool flag to require use of IPV4 address in case of multiple address family resolution by a DNS.
- `<prefix>_forceipv6` (`False`) bool flag to require use of IPV6 address in case of multiple address family resolution by a DNS.

3.4.2 Datagram transport class

Module `oscudpmc` manage transport for datagram protocols over IP : UDP, multicast, broadcast, etc. All these protocols share the same `UdpMcChannel` transport class, multicast and broadcast simply being enabled via options. An `UdpMcChannel` object can only act as a server or as a client, not both.

Datagram channel options

As such transport channels can be used either as a server (reader, receiving OSC packets) or as a client (writer, sending OSC packets), some options have read or write prefix and are only considered when using channel accordingly.

Read options

- `udpread_host` see "Network address options", above. This is the address where the socket is bound for reading.
- `udpread_port` see "Network address options", above. This is the port where the socket is bound for reading.
- `udpread_forceipv4` see "Network address options", above.
- `udpread_forceipv6` see "Network address options", above.
- `udpread_dontcache` (`False`) bool flag to not cache data in case of DNS resolution. By default resolved DNS addresses are cached in the application.
- `udpread_reuseaddr` (`True`) bool flag to enable ioctl settings for reuse of socket address.
- `udpread_nonblocking` (`True`) bool flag to enable non-blocking on the socket.
- `udpread_identusedns` (`False`) bool flag to translate address to DNS name using `oscnettools` DNS addresses cache.
- `udpread_identfields` (2) int count of fields of remote address identification to use for source identification. Use 0 for all fields. Default to 2 for (hostname, port) even with IPV6.
- `udpread_asstream` (`False`) bool flag to process UDP packets with stream-based methods, to manage rebuild of OSC packets from multiple UDP reads. Bad idea - but if you need it, don't miss to set up options like `read_withslip`, `read_withheader`...

Write options

- `udpwrite_host` see "Network address options", above. This is the address where the socket will send written packets.
- `udpwrite_port` see "Network address options", above. This is the port where the socket will send written packets.
- `udpwrite_outport` (0) int number of port to bind the socket locally. Default to 0 for auto-select.
- `udpwrite_forceipv4` see "Network address options", above.
- `udpwrite_forceipv6` see "Network address options", above.
- `udpwrite_dontcache` (`False`) bool flag to not cache data in case of DNS resolution. By default resolved DNS addresses are cached in the application.
- `udpread_reuseaddr` (`True`) bool flag to enable ioctl settings for reuse of socket address.
- `udpwrite_ttl` (`None`) - int time to leave counter for packets, also used for multicast hops. Default leave OS default settings.
- `udpwrite_nonblocking` (`True`) bool flag to enable non-blocking on the socket.

Multicast & Broadcast options

- `mcast_enabled` (`False`) bool flag to enable multicast. If True, the udpwrite_host must be a multicast group, and its a good idea to set udpwrite_ttl to 1 (or more if need to reach furthest networks).
- `mcast_loop` (`None`) bool flag to enable/disable looped back multicast packets to host. Normally enabled by default at the OS level. Default to None (don't modify OS settings).
- `bcast_enabled` (`False`) bool flag to enable broadcast. If True, the udpwrite_host must be a network broadcast address, and its a good idea to set `udpwrite_ttl` to 1 (or more if need to reach furthest networks).

3.4.3 Stream transport class

Network stream based transport using TCP is defined in module osctcp. It uses the class `TcpChannel` to manage connection and to transmissions.

TCP

- `tcp_consocket` (`None`) socket specified when creating a TcpChannel just after a connection, to manage communications with peer.
- `tcp_consocketspec` (`None`) tuple specifying socket specs.
- `tcp_host` see "Network address options", above. For a TCP server, you may generally use `"*"` here to require a server listening on all networks. For a TCP client, just specify the server host.
- `udpread_port` see "Network address options", above.
- `udpread_forceipv4` see "Network address options", above.
- `udpread_forceipv6` see "Network address options", above.
- `tcp_reuseaddr` (`True`) bool flag to enable ioctl settings for reuse of socket address.

3.5 User Helpers

These modules are described in the user documentation - we will not describe their interface.

3.5.1 as_eventloop

3.5.2 as_allthreads

3.5.3 as_comthreads

3.6 Action Handlers

These are action verbs which can be associated, at the transport channel level, to locally dispatched OSC messages or to direct callback functions to have special processing in some conditions.

3.6.1 Generic

These action handlers apply to all channel kind.

- **activating** — The channel is being activated (ie. system access via open() or like). No action parameter.
- **activated** — The channel has been activated. No action parameter.
- **deactivating** — The channel is being deactivated (ie. stop system access via close() or like). No action parameter.
- **deactivated** — The channel has been deactivated.
- **scheduling** — The channel is being scheduled (ie. begin monitoring of I/O on the underlying layers). No action parameter.
- **scheduled** — The channel has been scheduled. No action parameter.
- **unscheduling** — The channel is being unscheduled. No action parameter.
- **unscheduled** — The channel has been unscheduled. No action parameter.

TODO: Add handlers to get packetoption structures from the channel.

- **encodepacket** — A packet must be encoded to send via the channel.

  Two action parameters, the OSC source packet to encode and the packet options for processing. The handler must be a direct callback (as the second parameter is not valid for sending via OSC messages).

  If the handler return None, the processing of the packet continue (standard encoding, then sending).

  If the handler return (None, None), the processing of the packet stop - we consider that the handler manage itself the packet transmission to the channel.

  If the handler return (rawoscdata, packet option), they are used to transmit the raw packet via the channel, and standard encoding is not used.

- **decodepacket** — A packet coming from a transport channel must be decoded.

  Two action parameters, the raw OSC packet data to decode and the packet options for processing. The handler must be a direct callback (as the second parameter is not valid for sending via OSC messages).

  If the handler return None, the processing of the packet continue (standard decoding, then queue for dispatching).

  If the handler return (None, None), the processing of the packet stop - we consider that the handler manage itself the packet transmission to the dispatcher.

  If the handler return (packet, packet option), they are used to queue the packet for dispatching, and standard decoding is not used.

3.6.2 UDP

- **bound** — The UDP socket has been bound to a port, waiting for writing or reading. Action parameter is the port number.

3.6.3 TCP

- **conreq** — A connexion request has been received and a transport channel will be created to manage communications on the corresponding socket. A callback trigged on this handler can raise an exception to cancel the establishment of TCP communications. If a callback method

Two parameters: (address, sockfileno). The remote network address as a tuple (maybe more than two items with IPV6) and the socket file number as an integer.

- **connected** — A connexion has been established with a remote pair.

Python Module Index

o

Index