18.5.4. Transports and protocols (callback based API)

Source code: Lib/asyncio/transports.py

Source code: Lib/asyncio/protocols.py

18.5.4.1. Transports

Transports are classes provided by `asyncio` in order to abstract various kinds of communication channels. You generally won't instantiate a transport yourself; instead, you will call an `AbstractEventLoop` method which will create the transport and try to initiate the underlying communication channel, calling you back when it succeeds.

Once the communication channel is established, a transport is always paired with a protocol instance. The protocol can then call the transport's methods for various purposes.

`asyncio` currently implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport's kind.

The transport classes are not thread safe.

Changed in version 3.6: The socket option `TCP_NODELAY` is now set by default.

18.5.4.1.1. BaseTransport

class `asyncio.`**`BaseTransport`**

Base class for transports.

`close`()

Close the transport. If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's `connection_lost()` method will be called with `None` as its argument.

`is_closing`()

Return `True` if the transport is closing or is closed.

New in version 3.5.1.

get_extra_info(*name*, *default=None*)

Return optional transport information. *name* is a string representing the piece of transport-specific information to get, *default* is the value to return if the information doesn't exist.

This method allows transport implementations to easily expose channel-specific information.

- socket:
  - `'peername'`: the remote address to which the socket is connected, result of `socket.socket.getpeername()` (`None` on error)
  - `'socket'`: `socket.socket` instance
  - `'sockname'`: the socket's own address, result of `socket.socket.getsockname()`
- SSL socket:
  - `'compression'`: the compression algorithm being used as a string, or `None` if the connection isn't compressed; result of `ssl.SSLSocket.compression()`
  - `'cipher'`: a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of `ssl.SSLSocket.cipher()`
  - `'peercert'`: peer certificate; result of `ssl.SSLSocket.getpeercert()`
  - `'sslcontext'`: `ssl.SSLContext` instance
  - `'ssl_object'`: `ssl.SSLObject` or `ssl.SSLSocket` instance
- pipe:
  - `'pipe'`: pipe object
- subprocess:
  - `'subprocess'`: `subprocess.Popen` instance

set_protocol(*protocol*)

Set a new protocol. Switching protocol should only be done when both protocols are documented to support the switch.

New in version 3.5.3.

get_protocol()

Return the current protocol.

New in version 3.5.3.

Changed in version 3.5.1: `'ssl_object'` info was added to SSL sockets.

18.5.4.1.2. ReadTransport

class `asyncio.`**`ReadTransport`**

Interface for read-only transports.

`pause_reading`()

Pause the receiving end of the transport. No data will be passed to the protocol's `data_received()` method until `resume_reading()` is called.

Changed in version 3.6.7: The method is idempotent, i.e. it can be called when the transport is already paused or closed.

`resume_reading`()

Resume the receiving end. The protocol's `data_received()` method will be called once again if some data is available for reading.

Changed in version 3.6.7: The method is idempotent, i.e. it can be called when the transport is already reading.

18.5.4.1.3. WriteTransport

class `asyncio.`**`WriteTransport`**

Interface for write-only transports.

`abort`()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's `connection_lost()` method will eventually be called with `None` as its argument.

`can_write_eof`()

Return `True` if the transport supports `write_eof()`, `False` if not.

`get_write_buffer_size`()

Return the current size of the output buffer used by the transport.

`get_write_buffer_limits`()

Get the *high*- and *low*-water limits for write flow control. Return a tuple `(low, high)` where *low* and *high* are positive number of bytes.

Use `set_write_buffer_limits()` to set the limits.

New in version 3.4.2.

set_write_buffer_limits(*high=None*, *low=None*)

Set the *high*- and *low*-water limits for write flow control.

These two values (measured in number of bytes) control when the protocol's `pause_writing()` and `resume_writing()` methods are called. If specified, the low-water limit must be less than or equal to the high-water limit. Neither *high* nor *low* can be negative.

`pause_writing()` is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, `resume_writing()` is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high-water limit is given, the low-water limit defaults to an implementation-specific value less than or equal to the high-water limit. Setting *high* to zero forces *low* to zero as well, and causes `pause_writing()` to be called whenever the buffer becomes non-empty. Setting *low* to zero causes `resume_writing()` to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use `get_write_buffer_limits()` to get the limits.

write(*data*)

Write some *data* bytes to the transport.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

writelines(*list_of_data*)

Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling `write()` on each element yielded by the iterable, but may be implemented more efficiently.

write_eof()

Close the write end of the transport after flushing buffered data. Data may still be received.

This method can raise `NotImplementedError` if the transport (e.g. SSL) doesn't support half-closes.

18.5.4.1.4. DatagramTransport

`DatagramTransport.` **sendto**(*data*, *addr=None*)

Send the *data* bytes to the remote peer given by *addr* (a transport-dependent target address). If *addr* is `None`, the data is sent to the target address given on transport creation.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

`DatagramTransport.` **`abort`**()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's `connection_lost()` method will eventually be called with `None` as its argument.

18.5.4.1.5. BaseSubprocessTransport

class `asyncio.` **`BaseSubprocessTransport`**

`get_pid`()

Return the subprocess process id as an integer.

`get_pipe_transport`(*fd*)

Return the transport for the communication pipe corresponding to the integer file descriptor *fd*:

- `0`: readable streaming transport of the standard input (*stdin*), or `None` if the subprocess was not created with `stdin=PIPE`
- `1`: writable streaming transport of the standard output (*stdout*), or `None` if the subprocess was not created with `stdout=PIPE`
- `2`: writable streaming transport of the standard error (*stderr*), or `None` if the subprocess was not created with `stderr=PIPE`
- other *fd*: `None`

`get_returncode`()

Return the subprocess returncode as an integer or `None` if it hasn't returned, similarly to the `subprocess.Popen.returncode` attribute.

`kill`()

Kill the subprocess, as in `subprocess.Popen.kill()`.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for `terminate()`.

`send_signal`(*signal*)

Send the *signal* number to the subprocess, as in `subprocess.Popen.send_signal()`.

terminate()

Ask the subprocess to stop, as in `subprocess.Popen.terminate()`. This method is an alias for the `close()` method.

On POSIX systems, this method sends SIGTERM to the subprocess. On Windows, the Windows API function TerminateProcess() is called to stop the subprocess.

close()

Ask the subprocess to stop by calling the `terminate()` method if the subprocess hasn't returned yet, and close transports of all pipes (*stdin*, *stdout* and *stderr*).

18.5.4.2. Protocols

`asyncio` provides base classes that you can subclass to implement your network protocols. Those classes are used in conjunction with transports (see below): the protocol parses incoming data and asks for the writing of outgoing data, while the transport is responsible for the actual I/O and buffering.

When subclassing a protocol class, it is recommended you override certain methods. Those methods are callbacks: they will be called by the transport on certain events (for example when some data is received); you shouldn't call them yourself, unless you are implementing a transport.

> **Note:** All callbacks have default implementations, which are empty. Therefore, you only need to implement the callbacks for the events in which you are interested.

18.5.4.2.1. Protocol classes

class `asyncio.`**Protocol**

The base class for implementing streaming protocols (for use with e.g. TCP and SSL transports).

class `asyncio.`**DatagramProtocol**

The base class for implementing datagram protocols (for use with e.g. UDP transports).

class `asyncio.`**SubprocessProtocol**

The base class for implementing protocols communicating with child processes (through a set of unidirectional pipes).

18.5.4.2.2. Connection callbacks

These callbacks may be called on `Protocol`, `DatagramProtocol` and `SubprocessProtocol` instances:

`BaseProtocol.`**`connection_made`**(*transport*)
Called when a connection is made.

The *transport* argument is the transport representing the connection. You are responsible for storing it somewhere (e.g. as an attribute) if you need to.

`BaseProtocol.`**`connection_lost`**(*exc*)
Called when the connection is lost or closed.

The argument is either an exception object or `None`. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

`connection_made()` and `connection_lost()` are called exactly once per successful connection. All other callbacks will be called between those two methods, which allows for easier resource management in your protocol implementation.

The following callbacks may be called only on `SubprocessProtocol` instances:

`SubprocessProtocol.`**`pipe_data_received`**(*fd*, *data*)
Called when the child process writes data into its stdout or stderr pipe. *fd* is the integer file descriptor of the pipe. *data* is a non-empty bytes object containing the data.

`SubprocessProtocol.`**`pipe_connection_lost`**(*fd*, *exc*)
Called when one of the pipes communicating with the child process is closed. *fd* is the integer file descriptor that was closed.

`SubprocessProtocol.`**`process_exited`**()
Called when the child process has exited.

18.5.4.2.3. Streaming protocols

The following callbacks are called on `Protocol` instances:

`Protocol.`**`data_received`**(*data*)
Called when some data is received. *data* is a non-empty bytes object containing the incoming data.

> **Note:** Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible enough. However, data is always received in the correct order.

`Protocol.`**`eof_received`**()

Called when the other end signals it won't send any more data (for example by calling `write_eof()`, if the other end also uses asyncio).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, closing the transport is up to the protocol. Since the default implementation returns `None`, it implicitly closes the connection.

> **Note:** Some transports such as SSL don't support half-closed connections, in which case returning true from this method will not prevent closing the connection.

`data_received()` can be called an arbitrary number of times during a connection. However, `eof_received()` is called at most once and, if called, `data_received()` won't be called after it.

State machine:

> start -> `connection_made()` [-> `data_received()` *] [-> `eof_received()` ?] -> `connection_lost()` -> end

18.5.4.2.4. Datagram protocols

The following callbacks are called on `DatagramProtocol` instances.

`DatagramProtocol.`**`datagram_received`**(*data*, *addr*)

Called when a datagram is received. *data* is a bytes object containing the incoming data. *addr* is the address of the peer sending the data; the exact format depends on the transport.

`DatagramProtocol.`**`error_received`**(*exc*)

Called when a previous send or receive operation raises an `OSError`. *exc* is the `OSError` instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram couldn't be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

18.5.4.2.5. Flow control callbacks

These callbacks may be called on `Protocol`, `DatagramProtocol` and `SubprocessProtocol` instances:

`BaseProtocol.`**`pause_writing`**()
: Called when the transport's buffer goes over the high-water mark.

`BaseProtocol.`**`resume_writing`**()
: Called when the transport's buffer drains below the low-water mark.

`pause_writing()` and `resume_writing()` calls are paired – `pause_writing()` is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually `resume_writing()` is called once when the buffer size reaches the low-water mark.

Note: If the buffer size equals the high-water mark, `pause_writing()` is not called – it must go strictly over. Conversely, `resume_writing()` is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

Note: On BSD systems (OS X, FreeBSD, etc.) flow control is not supported for `DatagramProtocol`, because send failures caused by writing too many packets cannot be detected easily. The socket always appears 'ready' and excess packets are dropped; an `OSError` with errno set to `errno.ENOBUFS` may or may not be raised; if it is raised, it will be reported to `DatagramProtocol.error_received()` but otherwise ignored.

18.5.4.2.6. Coroutines and protocols

Coroutines can be scheduled in a protocol method using `ensure_future()`, but there is no guarantee made about the execution order. Protocols are not aware of coroutines created in protocol methods and so will not wait for them.

To have a reliable execution order, use stream objects in a coroutine with `yield from`. For example, the `StreamWriter.drain()` coroutine can be used to wait until the write buffer is flushed.

18.5.4.3. Protocol examples

18.5.4.3.1. TCP echo client protocol

TCP echo client using the `AbstractEventLoop.create_connection()` method, send data and wait until the connection is closed:

```
import asyncio

class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        print('Stop the event loop')
        self.loop.stop()

loop = asyncio.get_event_loop()
message = 'Hello World!'
coro = loop.create_connection(lambda: EchoClientProtocol(message, loop
                              '127.0.0.1', 8888)
loop.run_until_complete(coro)
loop.run_forever()
loop.close()
```

The event loop is running twice. The `run_until_complete()` method is preferred in this short example to raise an exception if the server is not listening, instead of having to write a short coroutine to handle the exception and stop the running loop. At `run_until_complete()` exit, the loop is no longer running, so there is no need to stop the loop in case of an error.

See also: The TCP echo client using streams example uses the `asyncio.open_connection()` function.

18.5.4.3.2. TCP echo server protocol

TCP echo server using the `AbstractEventLoop.create_server()` method, send back received data and close the connection:

```
import asyncio

class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()

loop = asyncio.get_event_loop()
# Each client connection will create a new protocol instance
coro = loop.create_server(EchoServerClientProtocol, '127.0.0.1', 8888)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

`Transport.close()` can be called immediately after `WriteTransport.write()` even if data are not sent yet on the socket: both methods are asynchronous. `yield from` is not needed because these transport methods are not coroutines.

See also: The TCP echo server using streams example uses the `asyncio.start_server()` function.

18.5.4.3.3. UDP echo client protocol

UDP echo client using the `AbstractEventLoop.create_datagram_endpoint()` method, send data and close the transport when we received the answer:

```
import asyncio

class EchoClientProtocol:
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Socket closed, stop the event loop")
        loop = asyncio.get_event_loop()
        loop.stop()

loop = asyncio.get_event_loop()
message = "Hello World!"
connect = loop.create_datagram_endpoint(
    lambda: EchoClientProtocol(message, loop),
    remote_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(connect)
loop.run_forever()
transport.close()
loop.close()
```

18.5.4.3.4. UDP echo server protocol

UDP echo server using the `AbstractEventLoop.create_datagram_endpoint()` method, send back received data:

```
import asyncio

class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
```

```
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)

loop = asyncio.get_event_loop()
print("Starting UDP server")
# One protocol instance will be created to serve all client requests
listen = loop.create_datagram_endpoint(
    EchoServerProtocol, local_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(listen)

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

transport.close()
loop.close()
```

18.5.4.3.5. Register an open socket to wait for data using a protocol

Wait until a socket receives data using the `AbstractEventLoop.create_connection()` method with a protocol, and then close the event loop

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

# Create a pair of connected sockets
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

class MyProtocol(asyncio.Protocol):
    transport = None

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport (it will call connection_lo
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed, stop the event loop
```

```
        loop.stop()

# Register the socket to wait for data
connect_coro = loop.create_connection(MyProtocol, sock=rsock)
transport, protocol = loop.run_until_complete(connect_coro)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()
```

See also: The watch a file descriptor for read events example uses the low-level `AbstractEventLoop.add_reader()` method to register the file descriptor of a socket.

The register an open socket to wait for data using streams example uses high-level streams created by the `open_connection()` function in a coroutine.