Ohne I/O Documentation

Release 0.9

Antoine Catton

October 09, 2016

Contents

Ohne I/O (or `ohneio`) is an utility library to write network protocol parsers without depending on any I/O system.

This allows the python community to then use these protocol parsers with blocking raw sockets, asyncio, twisted, gevent, tornado, and/or threads/multiprocesses.

CHAPTER 1

Introduction

1.1 FAQ

1.1.1 Why not asyncio ?

asyncio is a great library, and Ohne I/O can be used in combination with asyncio, but it can also be used in combination with raw sockets and/or any network library.

Ohne I/O does not intend to compete in *any way* with asyncio, it even intends to be used in combination with asyncio. (but it is not limited to asyncio)

The power of `ohneio` resides in the fact that **it does not produce any I/O**. It only buffers byte streams, and allows developers to write simple stream parser like they would write a coroutine.

This is how you would use an `ohneio` protocol:

```
parser = protocol()
parser.send(bytes_to_send)
received_bytes = parser.read()
```

Because `ohneio` can be considered as connection without I/O, this documentation will referred to them with the variable `conn`.

Writing an `ohneio` protocol parser is better than writing an asyncio protocol library, in the sense that your protocol could be then used by anybody with any library.

Ohne I/O relies on the concept of sans-io, which is the concept of writing generic protocol parser, easily testable.

1.1.2 How does it work internally?

`ohneio` buffers data coming in or out. Thanks to generator functions, it pauses protocol parsers execution until new data is fed in or read from the parser.

1.1.3 Why not use manual buffers?

`ohneio` kinda borrows from parser combinators, it allows you to write simple functions without passing objects around. And combine these functions without managing any state.

`ohneio` also provides some primitives to wait for a certain amount of data, or wait for more data.

Writing simple protocol parser, could very quickly lead to either spaghetti code, badly abstracted code. `ohneio` tries to avoid this.

1.2 Getting started

To get started we'll write a simple line echo server. This protocol echoes each line sent to the server. A line is defined by a sequence of bytes terminated by the bytes `0x0A`.

1.2.1 Writing a protocol

```
import ohneio


NEW_LINE = b'\n'


def wait_for(s):
    """Wait for a certain string to be available in the buffer."""
    while True:
        data = yield from ohneio.peek()
        pos = data.find(s)
        if pos > 0:
            return pos
        yield from ohneio.wait()


def read_upto(s):
    """Read data up to the specified string in the buffer.

    If this string is not available in the buffer, this waits for the string to be available.
    """
    pos = yield from wait_for(s)
    data = yield from ohneio.read(pos + len(s))
    return data


@ohneio.protocol
def echo():
    while True:
        line = yield from read_upto(NEW_LINE)
        yield from ohneio.write(line)
```

1.2.2 Testing the protocol

Because the protocol doesn't produce any I/O, you can directly simulates in which order, and which segments of data will be read/buffered. And what should be send back.

```
import pytest


@pytest.fixture
def conn():
    return echo()


@pytest.mark.parametrize('input_, expected_output', [
    (b"Hello World", b""),
    (b"Hello World\n", b"Hello World\n"),
```

```
    (b"Hello World\nAnd", b"Hello World\n"),
    (b"Hello World\nAnd the universe\n", b"Hello World\nAnd the universe\n"),
])
def test_only_echo_complete_lines(conn, input_, expected_output):
    conn.send(input_)
    assert conn.read() == expected_output


def test_buffers_segments(conn):
    conn.send(b"Hello ")
    assert conn.read() == b''
    conn.send(b"World\n")
    assert conn.read() == b"Hello World\n"
```

1.2.3 Using a protocol

Now that you wrote your echo protocol, you need to make it use the network, you can use any network library you want:

Raw sockets

This example shows how to use Ohne I/O in combination with raw sockets and `select()`. This creates one instance of the `echo()` protocol per connection.

It then feeds data comming in the connection to the Ohne I/O protocol, then it feeds the output of the Ohne I/O protocol to the socket.

```
import contextlib
import select
import socket


BUFFER_SIZE = 1024


@contextlib.contextmanager
def create_server(host, port):
    for res in socket.getaddrinfo(host, port):
        af, socktype, proto, canonname, sa = res
        try:
            s = socket.socket(af, socktype, proto)
        except socket.error:
            continue
        try:
            s.bind(sa)
            s.listen(1)
            yield s
            return
        except socket.error:
            continue
        finally:
            s.close()
    raise RuntimeError("Couldn't create a connection")


def echo_server(host, port):
```

```
    connections = {}  # open connections: fileno -> (socket, protocol)
    with create_server(host, port) as server:
        while True:
            rlist, _, _ = select.select([server.fileno()] + list(connections.keys()), [], [])

            for fileno in rlist:
                if fileno == server.fileno():  # New connection
                    conn, _ = server.accept()
                    connections[conn.fileno()] = (conn, echo())
                else:  # Data comming in
                    sock, proto = connections[fileno]
                    data = sock.recv(BUFFER_SIZE)
                    if not data:  # Socket closed
                        del connections[fileno]
                        continue
                    proto.send(data)
                    data = proto.read()
                    if data:
                        sock.send(data)


if __name__ == '__main__':
    import sys
    echo_server(host=sys.argv[1], port=sys.argv[2])
```

gevent

This example shows how to use Ohne I/O in combination with gevent library:

```
import gevent
from gevent.server import StreamServer


BUFFER_SIZE = 1024


def echo_server(host, port):
    server = StreamServer((host, port), handle)
    try:
        server.serve_forever()
    finally:
        server.close()


def handle(socket, address):
    conn = echo()
    try:
        while True:
            data = socket.recv(BUFFER_SIZE)
            if not data:
                break
            conn.send(data)
            data = conn.read()
            if data:
                socket.send(data)
            gevent.sleep(0)  # Prevent one green thread from taking over
    finally:
```

```
        socket.close()


if __name__ == '__main__':
    import sys
    echo_server(host=sys.argv[1], port=int(sys.argv[2]))
```

asyncio

```
import asyncio


class EchoProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        self.transport = transport
        self.ohneio = echo()

    def data_received(self, data):
        self.ohneio.send(data)
        output = self.ohneio.read()
        if output:
            self.transport.write(output)



if __name__ == '__main__':
    import sys
    loop = asyncio.get_event_loop()
    coro = loop.create_server(EchoProtocol, host=sys.argv[1], port=int(sys.argv[2]))
    server = loop.run_until_complete(coro)
    try:
        loop.run_forever()
    finally:
        server.close()
        loop.run_until_complete(server.wait_closed())
        loop.close()
```

CHAPTER 2

API Documentation

CHAPTER 3

Indices and tables

- genindex
- modindex
- search