Asyncio and You

You Won’t Believe What Happens Next

Justus Perlwitz

Freelance Software Developer

Twitter, Github:

@justuswilhelm

Asyncio?!

A set of best practices

- Event Loops
- Coroutines
- Networking Code
- New Syntax

You Won’t Believe How Many Implementations Exist

- **Eventlet**
- **Greenlet**
- **Gevent**
- **Multitask**
- **Shrapnel**
- **Tornado**
- **asyncio**
- **… and many more**

What is in a good async library?

Event Loop

```
def initialize():
    ...

def get_next_message():
    ...

def process_message(message):
    ...
```

```
def main():
    initialize()
    while True:
        message = get_next_message()
        process_message(message)
```

Event Loops on *NIX

Instead of `get_message` we use

- `select` (POSIX)
- `epoll` (Linux)
- `kqueue` (*BSD, OS X)
- `IOCP` (Windows, Solaris)

Event Loop Libraries

OS-Independent Wrapper:

- libuv (Node.js)
- libevent (Chrome, ntpd, …)

… and in Python?

```
import selectors

def initialize():
    sock = socket.socket()
    ...
    selector.register(sock, selectors.
EVENT_READ, callback_method)

def callback_method():
    ...
```

```
def main():
    while True:
        events = sel.select()
        for key, mask in events:
            callback = key.data
            callback(key.fileobj, mask)
```

Coroutine

A method that has the following properties

- Multiple entry and exit points
- Suspendable Execution
- Thread safe (hopefully) when suspended

Why We Need Coroutines (1)

```
import selectors
sel = selectors.DefaultSelector()

def initialize():
    sock = socket.socket()
    sock.setblocking(False)
    sock.connect(('xkcd.com', 80))
    sel.register(sock.fileno(), selectors.
EVENT_WRITE, connected)
```

```
def main():
    while True:
        events = sel.select()
        for key, mask in events:
            callback = key.data
            callback(key.fileobj, mask)
```

Why We Need Coroutines (2)

```
def connected(sock, mask):
    sel.unregister()
    request = 'GET / HTTP 1.0\r\nHost:
xkcd.com\r\n\r\n'
    sock.send(request)
    sel.register(sock, selector.
EVENT_READ, read)
```

```
def read(sock, mask):
    ... # process request
    sel.unregister()
```

Callback Hell

Coroutines offer a clean solution

```
def connected(sock, mask):
    request = 'GET / HTTP 1.0\r\nHost: xkcd.com\r\n\r\n'
    response = yield from ???.send(request)
    content = yield from ???.read(response)
```

Combining event loops and coroutines: Introducing asyncio

asyncio (PEP 3156)

Formerly known as tulip, offers an

- Event Loop,
- Transport and Protocol Abstractions,
- Futures, Delayed Calls, Coroutines,
- Synchronization Primitives, and
- Thread pools (for blocking I/O calls).

Built into the Python stdlib in Python 3.4

async/await (PEP 0492)

While Python 3.4 introduced asyncio, Python 3.5 introduced async/await syntax:

```
async def example():
    response = await get('http://www.google.com')


import asyncio
@asyncio.coroutine
def example():
    response = yield from get('http://www.google.com')
```

High Level Coroutine Example

```python
from asyncio import get_event_loop
from aiohttp import get

async def example():
    response = await get('http://www.xkcd.com/')
    content = await response.text()
    return content[:15]

loop = get_event_loop()
result = loop.run_until_complete(example())
print(result)
```

Networking in Asyncio

Already, many highlevel libraries, such as

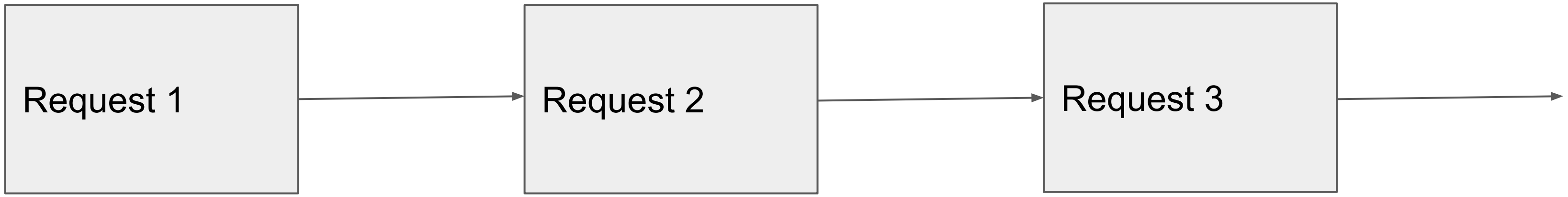
- aiohttp (HTTP Client/Server)
- aiopg (PostgreSQL)
- vase (HTTP Server),
- … (check out http://asyncio.org)

many more can be easily developed!

A more sophisticated example
or
How to build a Load Tester

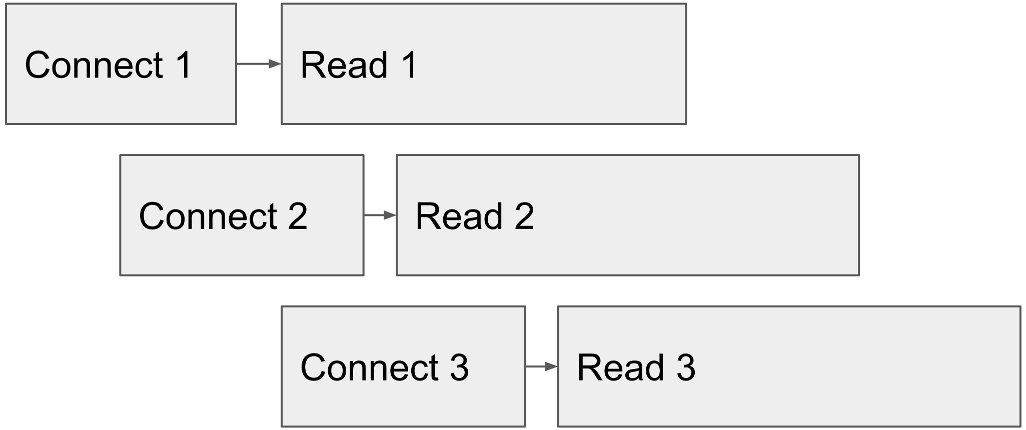
Load Testing is not possible with blocking I/O

We want to fire X concurrent requests to simulate realistic server loads



Only asynchronous I/O can solve this

Allows interleaving of requests



Works even for long-running tasks

We only want X concurrent requests

```python
from asyncio import Semaphore
MAX_CONCURRENT = 5
request_semaphore = Semaphore(MAX_CONCURRENT)
```

Main coroutine

```
async def client_request(url):
    start = time()
    async with request_semaphore:
        async with get(url) as response:
            await response.text()
    duration = time() - start
    return duration
```

Generating the Tasks

```
from asyncio import get_event_loop, wait
requests = 100
tasks = wait(list(gen_tasks()))
loop = get_event_loop()
```

Retrieving Results

```
done, _ = loop.run_until_complete(tasks)
average_task_duration = sum(map(lambda task: task.result(), done)) / requests
```

DEMO

Our very own aiohttp server!

Streaming Responses

```
@app.route('/large.csv')
def generate_large_csv():
    "Generate and serve a continuous stream of timestamps."
    def generate():
        for i in range(10):
            time.sleep(0.1)
            yield datetime.now().isoformat() + '\n'
    return Response(generate(), mimetype='text/csv')
```

DEMO

Now the same in asyncio

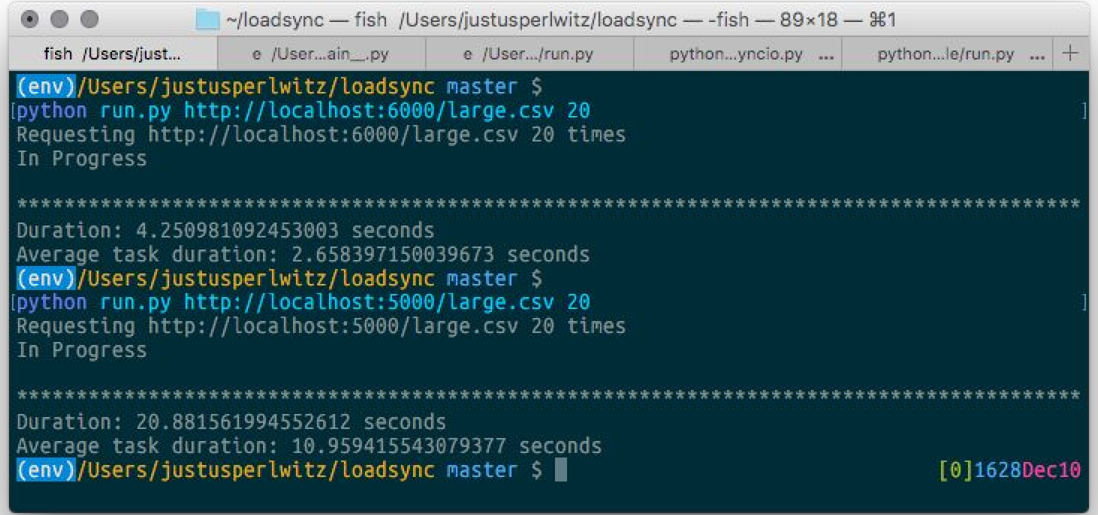
```
async def handle(request):
    print("GET /large.csv HTTP/1.1 200")
    response = web.StreamResponse()
    await response.prepare(request)
    for i in range(10):
        await sleep(0.1)
        response.write((datetime.now().isoformat() + '\n').encode())
        await response.drain()
    await response.write_eof()
    return response
```

DEMO

The Result

20 seconds (synchronous)

5 seconds (asynchronous)



```
(env)/Users/justusperlwitz/loadsync master $
python run.py http://localhost:6000/large.csv 20
Requesting http://localhost:6000/large.csv 20 times
In Progress

*****************************************************************************************
Duration: 4.250981092453003 seconds
Average task duration: 2.658397150039673 seconds
(env)/Users/justusperlwitz/loadsync master $
python run.py http://localhost:5000/large.csv 20
Requesting http://localhost:5000/large.csv 20 times
In Progress

*****************************************************************************************
Duration: 20.881561994552612 seconds
Average task duration: 10.959415543079377 seconds
(env)/Users/justusperlwitz/loadsync master $                                [0]1628Dec10
```

Bonus Round

Testing Asyncio (Spoiler: it’s messy)

Enable Debug Mode with

PYTHONASYNCIODEBUG=1

Use a test library such as

- asynctest
- pytest-asyncio (if you’re using pytest)

Summary

When to use Asyncio (and when not)?

Use asyncio, if you have

- IO-bound tasks,
- Multiple, similar tasks,
- Independently executable tasks,
- Producer-Consumer-Model

Reconsider, if you have

- CPU-bound tasks,
- complex task dependencies

Thanks!

CPU 1
CPU 2
CPU 4
CPU 0
iPad.