Making 1 million requests with python-aiohttp

Apr 22, 2016 - by Paweł Miech - about: asyncio, aiohttp, python

In this post I'd like to test limits of python aiohttp and check its performance in terms of requests per minute. Everyone knows that asynchronous code performs better when applied to network operations, but it's still interesting to check this assumption and understand how exactly it is better and why it's is better. I'm going to check it by trying to make 1 million requests with aiohttp client. How many requests per minute will aiohttp make? What kind of exceptions and crashes can you expect when you try to make such volume of requests with very primitive scripts? What are main gotchas that you need to think about when trying to make such volume of requests?

Hello asyncio/aiohttp

Async programming is not easy. It's not easy because using callbacks and thinking in terms of events and event handlers requires more effort than usual synchronous programming. But it is also difficult because asyncio is still relatively new and there are few blog posts, tutorials about it. Official docs are very terse and contain only basic examples. There are some Stack Overflow questions but not that many only 410 as of time of writing (compare with 2 585 questions tagged "twisted") There are couple of nice blog posts and articles about asyncio over there such as this, that, that or perhaps even this or this.

To make it easier let's start with the basics - simple HTTP hello world - just making GET and fetching one single HTTP response.

In synchronous world you just do:

```
import requests

def hello():
    return requests.get("http://httpbin.org/get")

print(hello())
```

How does that look in aiohttp?

```
#!/usr/local/bin/python3.5
import asyncio
from aiohttp import ClientSession

async def hello(url):
    async with ClientSession() as session:
        async with session.get(url) as response:
            response = await response.read()
            print(response)

loop = asyncio.get_event_loop()

loop.run_until_complete(hello("http://httpbin.org/headers"))
```

hmm looks like I had to write lots of code for such a basic task… There is “async def” and “async with” and two “awaits” here. It seems really confusing at first sight, let’s try to explain it then.

You make your function asynchronous by using async keyword before function definition and using await keyword. There are actually two asynchronous operations that our hello() function performs. First it fetches response asynchronously, then it reads response body in asynchronous manner.

Aiohttp recommends to use ClientSession as primary interface to make requests. ClientSession allows you to store cookies between requests and keeps objects that are common for all requests (event loop, connection and other things). Session needs to be closed after using it, and closing session is another asynchronous operation, this is why you need `async with` every time you deal with sessions.

After you open client session you can use it to make requests. This is where another asynchronous operation starts, downloading request. Just as in case of client sessions responses must be closed explicitly, and context manager’s `with` statement ensures it will be closed properly in all circumstances.

To start your program you need to run it in event loop, so you need to create instance of asyncio loop and put task into this loop.

It all does sound bit difficult but it’s not that complex and looks logical if you spend some time trying to understand it.

Fetch multiple urls

Now let’s try to do something more interesting, fetching multiple urls one after another. With synchronous code you would do just:

```
for url in urls:
    print(requests.get(url).text)
```

This is really quick and easy, async will not be that easy, so you should always consider if something more complex is actually necessary for your needs. If your app works nice with synchronous code maybe there is no need to bother with async code? If you do need to bother with async code here's how you do that. Our `hello()` async function stays the same but we need to wrap it in asyncio `Future` object and pass whole lists of Future objects as tasks to be executed in the loop.

```
loop = asyncio.get_event_loop()

tasks = []
# I'm using test server localhost, but you can use any url
url = "http://localhost:8080/{}"
for i in range(5):
    task = asyncio.ensure_future(hello(url.format(i)))
    tasks.append(task)
loop.run_until_complete(asyncio.wait(tasks))
```

Now let's say we want to collect all responses in one list and do some postprocessing on them. At the moment we're not keeping response body anywhere, we just print it, let's return this response, keep it in list, and print all responses at the end.

To collect bunch of responses you probably need to write something along the lines of:

```
#!/usr/local/bin/python3.5
import asyncio
from aiohttp import ClientSession

async def fetch(url, session):
    async with session.get(url) as response:
        return await response.read()

async def run(r):
    url = "http://localhost:8080/{}"
    tasks = []

    # Fetch all responses within one Client session,
    # keep connection alive for all requests.
    async with ClientSession() as session:
        for i in range(r):
            task = asyncio.ensure_future(fetch(url.format(i), session))
            tasks.append(task)
```

```
        responses = await asyncio.gather(*tasks)
        # you now have all response bodies in this variable
        print(responses)

def print_responses(result):
    print(result)

loop = asyncio.get_event_loop()
future = asyncio.ensure_future(run(4))
loop.run_until_complete(future)
```

Notice usage of `asyncio.gather()`, this collects bunch of Future objects in one place and waits for all of them to finish.

Common gotchas

Now let's simulate real process of learning and let's make mistake in above script and try to debug it, this should be really helpful for demonstration purposes.

This is how sample broken async function looks like:

```
# WARNING! BROKEN CODE DO NOT COPY PASTE
async def fetch(url):
    async with ClientSession() as session:
        async with session.get(url) as response:
            return response.read()
```

This code is broken, but it's not that easy to figure out why if you dont know much about asyncio. Even if you know Python well but you dont know asyncio or aiohttp well you'll be in trouble to figure out what happens.

What is output of above function?

It produces following output:

```
pawel@pawel-VPCEH390X ~/p/l/benchmarker> ./bench.py
[<generator object ClientResponse.read at 0x7fa68d465728>, <generator object ClientRes
```

What happens here? You expected to get response objects after all processing is done, but here you actually get bunch of generators, why is that?

It happens because as I've mentioned earlier `response.read()` is async operation, this means that it does not return result immediately, it just returns generator. This generator still

needs to be called and executed, and this does not happen by default, `yield from` in Python 3.4 and `await` in Python 3.5 were added exactly for this purpose: to actually iterate over generator function. Fix to above error is just adding await before `response.read()`.

```
    # async operation must be preceded by await
    return await response.read() # NOT: return response.read()
```

Let's break our code in some other way.

```
# WARNING! BROKEN CODE DO NOT COPY PASTE
async def run(r):
    url = "http://localhost:8080/{}"
    tasks = []
    for i in range(r):
        task = asyncio.ensure_future(fetch(url.format(i)))
        tasks.append(task)

    responses = asyncio.gather(*tasks)
    print(responses)
```

Again above code is broken but it's not easy to figure out why if you're just learning asyncio.

Above produces following output:

```
pawel@pawel-VPCEH390X ~/p/l/benchmarker> ./bench.py
<_GatheringFuture pending>
Task was destroyed but it is pending!
task: <Task pending coro=<fetch() running at ./bench.py:7> wait_for=<Future pending cb
Task was destroyed but it is pending!
task: <Task pending coro=<fetch() running at ./bench.py:7> wait_for=<Future pending cb
Task was destroyed but it is pending!
task: <Task pending coro=<fetch() running at ./bench.py:7> wait_for=<Future pending cb
Task was destroyed but it is pending!
task: <Task pending coro=<fetch() running at ./bench.py:7> wait_for=<Future pending cb
```

What happens here? If you examine your localhost logs you may see that requests are not reaching your server at all. Clearly no requests are performed. Print statement prints that responses variable contains `<_GatheringFuture pending>` object, and later it alerts that pending tasks were destroyed. Why is it happening? Again you forgot about await

faulty line is this

```
    responses = asyncio.gather(*tasks)
```

it should be:

```
    responses = await asyncio.gather(*tasks)
```

I guess main lesson from those mistakes is: always remember about using “await” if you’re actually awaiting something.

Sync vs Async

Finally time for some fun. Let’s check if async is really worth the hassle. What’s the difference in efficiency between asynchronous client and blocking client? How many requests per minute can I send with my async client?

With this questions in mind I set up simple (async) aiohttp server. My server is going to read full html text of Frankenstein by Marry Shelley. It will add random delays between responses. Some responses will have zero delay, and some will have maximum of 3 seconds delay. This should resemble real applications, few apps respond to all requests with same latency, usually latency differs from response to response.

Server code looks like this:

```
#!/usr/local/bin/python3.5
import asyncio
from datetime import datetime
from aiohttp import web
import random

# set seed to ensure async and sync client get same distribution of delay values
# and tests are fair
random.seed(1)

async def hello(request):
    name = request.match_info.get("name", "foo")
    n = datetime.now().isoformat()
    delay = random.randint(0, 3)
    await asyncio.sleep(delay)
    headers = {"content_type": "text/html", "delay": str(delay)}
    # opening file is not async here, so it may block, to improve
    # efficiency of this you can consider using asyncio Executors
    # that will delegate file operation to separate thread or process
    # and improve performance
    # https://docs.python.org/3/library/asyncio-eventloop.html#executor
    # https://pymotw.com/3/asyncio/executors.html
    with open("frank.html", "rb") as html_body:
        print("{}: {} delay: {}".format(n, request.path, delay))
```

```
        response = web.Response(body=html_body.read(), headers=headers)
    return response

app = web.Application()
app.router.add_route("GET", "/{name}", hello)
web.run_app(app)
```

Synchronous client looks like this:

```
import requests
r = 100

url = "http://localhost:8080/{}"
for i in range(r):
    res = requests.get(url.format(i))
    delay = res.headers.get("DELAY")
    d = res.headers.get("DATE")
    print("{}:{} delay {}".format(d, res.url, delay))
```

How long will it take to run this?

On my machine running above synchronous client took 2:45.54 minutes.

My async code looks just like above code samples above. How long will async client take?

On my machine it took 0:03.48 seconds.

It is interesting that it took exactly as long as longest delay from my server. If you look into messages printed by client script you can see how great async HTTP client is. Some responses had 0 delay but others got 3 seconds delay. In synchronous client they would be blocking and waiting, your machine would simply stay idle for this time. Async client does not waste time, when something is delayed it simply does something else, issues other requests or processes all other responses. You can see this clearly in logs, first there are responses with 0 delay, then after they arrrived you can see responses with 1 seconds delay, and so on until most delayed responses arrive.

Testing the limits

Now that we know our async client is better let's try to test its limits and try to crash our localhost. I'm going to reset server delays to zero now (so no more random.choice of delays) and just see how fast we can go.

I'm going to start with sending 1k async requests. I'm curious how many requests my client can handle.

```
> time python3 bench.py

2.68user 0.24system 0:07.14elapsed 40%CPU (0avgtext+0avgdata 53704maxresident)k
0inputs+0outputs (0major+14156minor)pagefaults 0swaps
```

So 1k requests take 7 seconds, pretty nice! How about 10k? Trying to make 10k requests unfortunately fails…

```
responses are <_GatheringFuture finished exception=ClientOSError(24, 'Cannot connect t
Traceback (most recent call last):
  File "/home/pawel/.local/lib/python3.5/site-packages/aiohttp/connector.py", line 581
  File "/usr/local/lib/python3.5/asyncio/base_events.py", line 651, in create_connecti
  File "/usr/local/lib/python3.5/asyncio/base_events.py", line 618, in create_connecti
  File "/usr/local/lib/python3.5/socket.py", line 134, in __init__
OSError: [Errno 24] Too many open files
```

That's bad, seems like I stumbled across 10k connections problem.

It says "too many open files", and probably refers to number of open sockets. Why does it call them files? Sockets are just file descriptors, operating systems limit number of open sockets allowed. How many files are too many? I checked with python resource module and it seems like it's around 1024. How can we bypass this? Primitive way is just increasing limit of open files. But this is probably not the good way to go. Much better way is just adding some synchronization in your client limiting number of concurrent requests it can process. I'm going to do this by adding `asyncio.Semaphore()` with max tasks of 1000.

Modified client code looks like this now:

```
# modified fetch function with semaphore
import random
import asyncio
from aiohttp import ClientSession

async def fetch(url, session):
    async with session.get(url) as response:
        delay = response.headers.get("DELAY")
        date = response.headers.get("DATE")
        print("{}:{} with delay {}".format(date, response.url, delay))
        return await response.read()


async def bound_fetch(sem, url, session):
    # Getter function with semaphore.
    async with sem:
        await fetch(url, session)
```

```
async def run(r):
    url = "http://localhost:8080/{}"
    tasks = []
    # create instance of Semaphore
    sem = asyncio.Semaphore(1000)

    # Create client session that will ensure we dont open new connection
    # per each request.
    async with ClientSession() as session:
        for i in range(r):
            # pass Semaphore and session to every GET request
            task = asyncio.ensure_future(bound_fetch(sem, url.format(i), session))
            tasks.append(task)

        responses = asyncio.gather(*tasks)
        await responses

number = 10000
loop = asyncio.get_event_loop()

future = asyncio.ensure_future(run(number))
loop.run_until_complete(future)
```

At this point I can process 10k urls. It takes 23 seconds and returns some exceptions but overall it's pretty nice!

How about 100 000? This really makes my computer work hard but suprisingly it works ok. Server turns out to be suprisingly stable although you can see that ram usage gets pretty high at this point, cpu usage is around 100% all the time. What I find interesting is that my server takes significantly less cpu than client. Here's snapshot of linux `ps` output.

```
pawel@pawel-VPCEH390X ~/p/l/benchmarker> ps ua | grep python

USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
pawel     2447 56.3  1.0 216124 64976 pts/9    Sl+  21:26   1:27 /usr/local/bin/python
pawel     2527  101  3.5 674732 212076 pts/0   Rl+  21:26   2:30 /usr/local/bin/python
```

Overall it took around 53 seconds to process.

```
53.86user 1.58system 0:55.53elapsed 99%CPU (0avgtext+0avgdata 419216maxresident)k
0inputs+0outputs (0major+110195minor)pagefaults 0swaps
```

Pretty powerful if you ask me.

Finally I'm going to try 1 million requests. I really hope my laptop is not going to explode when testing that.

1 000 000 requests finished in 9 minutes.

```
530.86user 13.81system 9:05.17elapsed 99%CPU (0avgtext+0avgdata 3811640maxresident)k
0inputs+0outputs (0major+942576minor)pagefaults 0swaps
```

It means average request per minute rate of 111 111. Impressive.

Epilogue

You can see that asynchronous HTTP clients can be pretty powerful. Performing 1 million requests from async client is not difficult, and the client performs really well in comparison to synchronous code.

I wonder how it compares to other languages and async frameworks? Perhaps in some future post I could compare Twisted Treq with aiohttp. There is also question how many concurrent requests can be issued by async libraries in other languages. E.g. what would be results of benchmarks for some Java async frameworks? Or C++ frameworks? Or some Rust HTTP clients?

EDITS (24/04/2016)

- improved code sample that uses Semaphore
- added comment about using executor when opening file
- added link to HN comment about EADDRNOTAVAIL exception

EDITS (10/09/2016)

Earlier version of this post contained problematic usage of ClientSession that caused client to crash. You can find this older version of article here. For more details about this issue see this GitHub ticket.

EDITS (08/11/2016)

Fixed minor bugs in code samples:

- removed useless positional argument 'loop' to run()
- added positional argument url to hello() async def
- added missing colon in requests sync code sample

Python programming, web, data science
pawelmhm
pawelmhm+github_blog@gmail.com

Blog about programming (Python) and occasionally about data analysis