Hi team, thank you for the constructive and motivating feedback. This gives me a glimpse of hope that I should not give up on this position! ☺

I just started deep-dive researching the world of asynchronous requests to better understand what is going under the hood. Although most of the materials I've come across were kind of lacking for me to connect the dots, then I came up with these 2 Part GitHub blogposts which kind of cleared the fogs a bit. I highly recommend them, in case you are interested in fun read:
https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-1
https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-2

Table of Contents

Feedback Review ***1***

Code & Case studies ***2***

Empirical Studies ***4***

Performance Benchmark ***9***

Along the way ***13***

Final notes ***13***

Feedback Review

After carefully reviewing your feedback, I believe most of the code had to be adjusted to address points on feedback. So, few points I've come up with to discuss the drawbacks of previous code and their improvements in new version:

- *Your server is waiting for all requests to finish before sending the first successful response. Beware that some requests can take a really long time and the "time" property in response does not correspond to the actual request duration.*

Apparently, the cause of this is **asyncio.gather()** function which waits to collect the given request results. As a result, the previous code was waiting for all the requests to be finished before sending the first successful response. But, as we want to process requests greedily as they are ready, we will use the loop over **asyncio.as_completed()** [https://bit.ly/as_compled]. As the name implies, it helps us to get the earliest next result among requests.

Also, in my last implementation, I was mistakenly considering time property in response from Exponea server as real time, and the code was working based on that time. In this new implementation, the code will work with time that is based on session rather than Exponea's response.

- "*Second and third requests are sent after the first request finishes instead of waiting 300ms since the request has been started.*"

The cause, again is **asyncio.gather()** which waits the first request to finish. To address this, the closest I could come up with is to use **asyncio.wait_for()** with **asyncio.shield()** [https://bit.ly/wait_for_shield]. More about this is below.

- *You mentioned the possibility to run tests but you forgot to push them.*

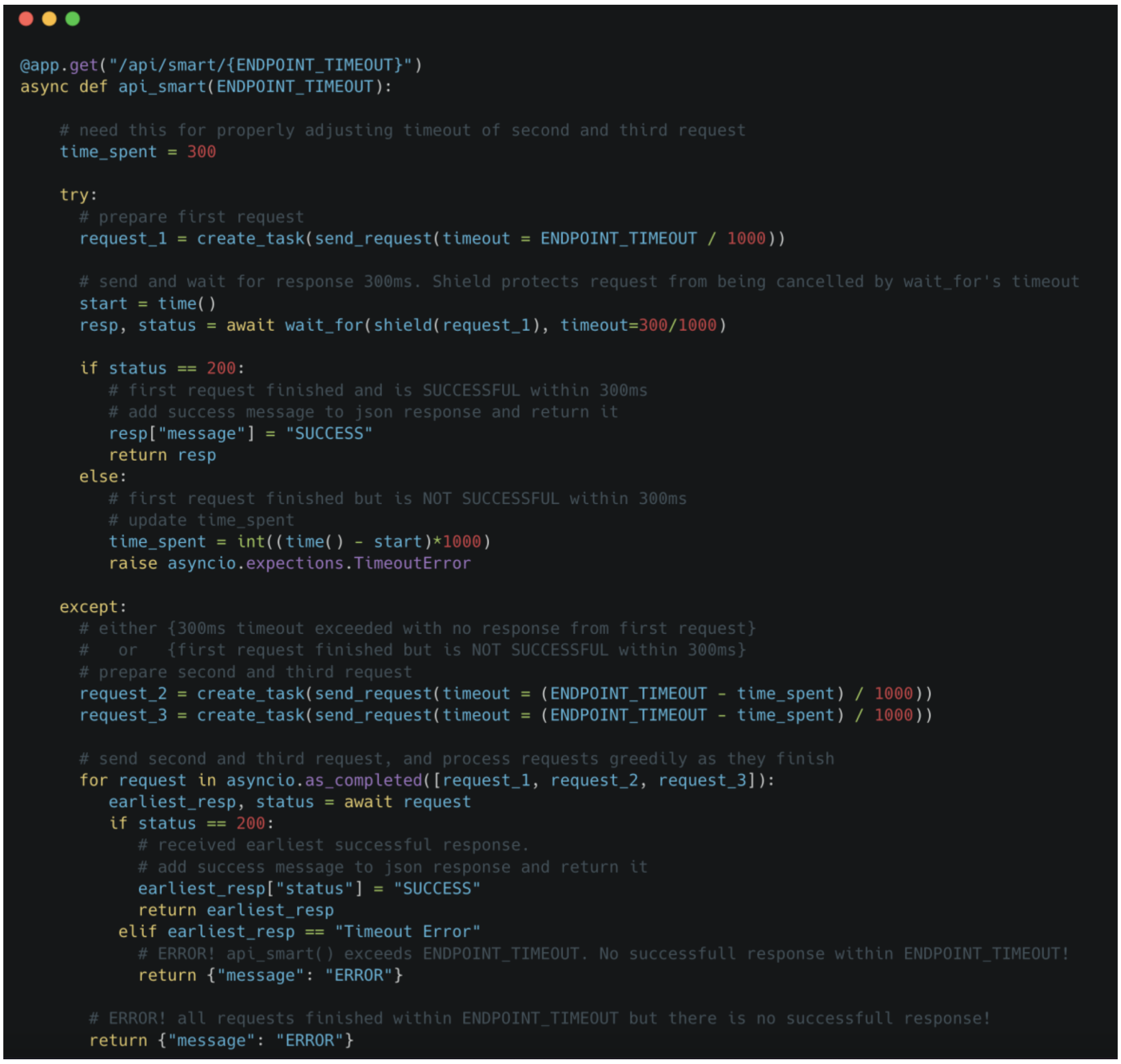
Added pytests into api_test.py script. The script runs tests and checks the possible cases.

- *We would love to see your discussion about the server's performance. Feel free to run a performance benchmark and share your results with us. How many concurrent requests can the server handle? How would you protect the server against being overloaded?*

I used Apache JMeter for performance benchmarking and discussed the server behavior at the end of this report.

Code & Case studies

First, let's explore the pseudo-code of our new approach. It is pseudo by means that I included the most essential parts for taking screenshot. Original code is just slightly different than this but they work the same way.

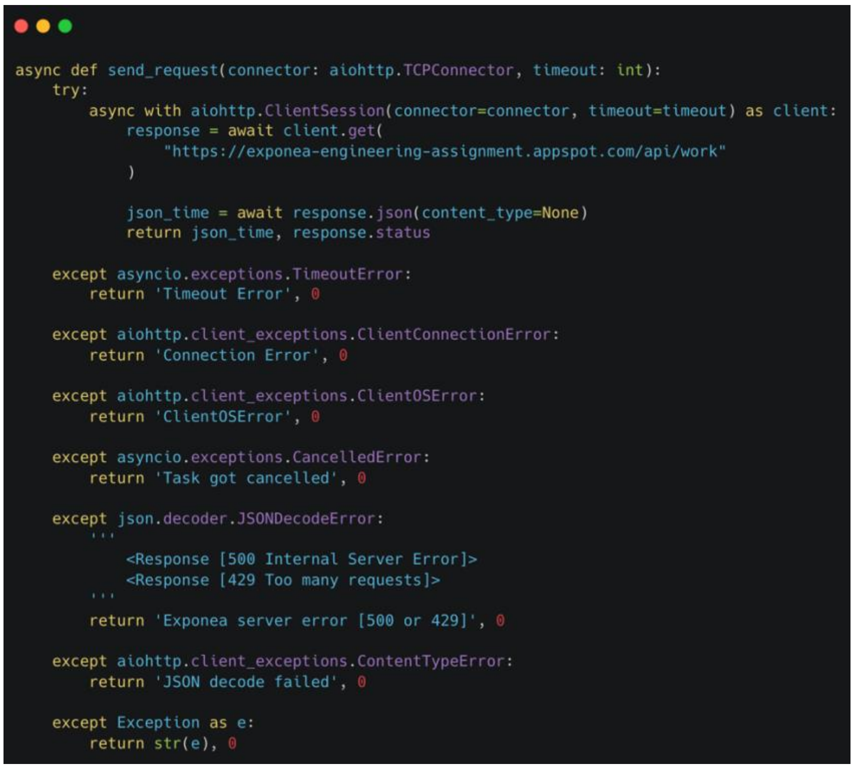


```
@app.get("/api/smart/{ENDPOINT_TIMEOUT}")
async def api_smart(ENDPOINT_TIMEOUT):

    # need this for properly adjusting timeout of second and third request
    time_spent = 300

    try:
      # prepare first request
      request_1 = create_task(send_request(timeout = ENDPOINT_TIMEOUT / 1000))

      # send and wait for response 300ms. Shield protects request from being cancelled by wait_for's timeout
      start = time()
      resp, status = await wait_for(shield(request_1), timeout=300/1000)

      if status == 200:
         # first request finished and is SUCCESSFUL within 300ms
         # add success message to json response and return it
         resp["message"] = "SUCCESS"
         return resp
      else:
         # first request finished but is NOT SUCCESSFUL within 300ms
         # update time_spent
         time_spent = int((time() - start)*1000)
         raise asyncio.expections.TimeoutError

    except:
      # either {300ms timeout exceeded with no response from first request}
      #    or   {first request finished but is NOT SUCCESSFUL within 300ms}
      # prepare second and third request
      request_2 = create_task(send_request(timeout = (ENDPOINT_TIMEOUT - time_spent) / 1000))
      request_3 = create_task(send_request(timeout = (ENDPOINT_TIMEOUT - time_spent) / 1000))

      # send second and third request, and process requests greedily as they finish
      for request in asyncio.as_completed([request_1, request_2, request_3]):
         earliest_resp, status = await request
         if status == 200:
            # received earliest successful response.
            # add success message to json response and return it
            earliest_resp["status"] = "SUCCESS"
            return earliest_resp
          elif earliest_resp == "Timeout Error"
            # ERROR! api_smart() exceeds ENDPOINT_TIMEOUT. No successfull response within ENDPOINT_TIMEOUT!
            return {"message": "ERROR"}

       # ERROR! all requests finished within ENDPOINT_TIMEOUT but there is no successfull response!
       return {"message": "ERROR"}
```

Note that, for the sake of chosen implementation way, api_smart() only handles TimeoutError exception which is thrown by either of below:

- **by async.wait_for() if it exceeds 300ms**
- **by manually raising it**

The reason for this is that, on the other side, send_request() can handle all exceptions that may occur during the sessions and returns the respective response and status of requests back to api_smart(). This helps api_smart() to evaluate the requests according to their responses and statuses, and handle exceptions caused by sessions.



```python
async def send_request(connector: aiohttp.TCPConnector, timeout: int):
    try:
        async with aiohttp.ClientSession(connector=connector, timeout=timeout) as client:
            response = await client.get(
                "https://exponea-engineering-assignment.appspot.com/api/work"
            )

            json_time = await response.json(content_type=None)
            return json_time, response.status

    except asyncio.exceptions.TimeoutError:
        return 'Timeout Error', 0

    except aiohttp.client_exceptions.ClientConnectionError:
        return 'Connection Error', 0

    except aiohttp.client_exceptions.ClientOSError:
        return 'ClientOSError', 0

    except asyncio.exceptions.CancelledError:
        return 'Task got cancelled', 0

    except json.decoder.JSONDecodeError:
        '''
            <Response [500 Internal Server Error]>
            <Response [429 Too many requests]>
        '''
        return 'Exponea server error [500 or 429]', 0

    except aiohttp.client_exceptions.ContentTypeError:
        return 'JSON decode failed', 0

    except Exception as e:
        return str(e), 0
```

Now, let's see how our server works. Once our API receives a GET request via endpoint, api_smart() starts with checking if endpoint received an integer and if it is below 300. If it is not integer, then we return message "Endpoint timeout parameter should be integer". If it is integer but is below 300, then we return message "Endpoint timeout parameter should be above 300". If it is integer and is above 300, then api_smart() continues with firing the first request with timeout=ENDPOINT_TIMEOUT and waiting for response 300ms:

1) If within 300ms, first request finishes and is successful within 300ms. (i.e., 200), then we return its response.
2) If within 300ms, first request finishes, but is not successful (i.e., !200), then we manually raise TimeoutError. Idea is that, as our first request failed, it is time to fire two other requests ***without waiting 300ms timeout***. Now, in order to handle the TimeoutError, code continues with "except" clause and fires two other requests and process all requests inside asyncio.as_completed() loop greedily as they finish and we return the earliest successful response. It is important to note that, although the first request is finished unsuccessfully, for the sake of chosen implementation way, we will still await its task during asyncio.as_completed() loop. Lets remember that its task is "finished task", and according to https://bit.ly/await_finished_task, awaiting the finished task returns its result immediately – so, we are okay. Now, if there is no any successful response out of second and third requests, loop will finish without returning, and we will continue with the next line which returns ERROR. On the other hand, if there is/are successful responses, the earliest one is returned during the loop.
3) If 300ms exceeds with no response from first request, wait_for() throws TimeoutError exception. Idea is that, we waited for first request's response for 300ms, but as it did not finish within 300ms, it is time to fire two other requests. Note that in this case, we are sure that first request did not finish within 300ms, because if it had finished within 300ms, it would be either case 1) or 2). It is also

important to note that we have to protect first request with asyncio.shield() in order for it not to be cancelled once wait_for() timeout occurs. Reason is that, even if 300ms timeout occurs, we still need the response of first request. Now, to handle the TimeoutError thrown by wait_for(), code continues with "except" clause. This is where we fire two other requests and process requests greedily as they finish using asyncio.as_completed() and return the earliest successful response. If there is no any successful response out of all three requests, loop will finish without returning, and we will continue with the next line which returns ERROR. On the other hand, if there is/are successful responses, the earliest one is returned during the loop.

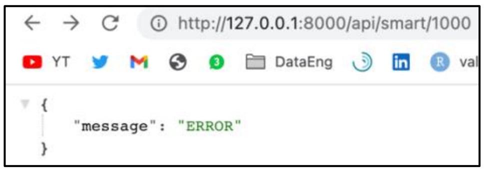
4) Last case is when during any of above cases, api_smart() execution time exceeds ENDPOINT_TIMEOUT. This is approximately addressed by allocating timeout for each request's session such that each session throws TimeoutError when api_smart() exceeds ENDPOINT_TIMEOUT. It is important to note that, this is an approximate implementation, and after studying the server's behavior, we see that sessions throw TimeoutError **almost at the same time with 10-30ms difference in-between**. Also, most of the time, first request's session is the earliest to throw TimeoutError. Please, refer to the code implementation for getting more insights on this topic. Finally, I also added **elif statement** into as_completed() loop which **returns ERROR** if any request's session threw TimeoutError – this is where we know if api_smart() execution time exceeds ENDPOINT_TIMEOUT.
   Moreover, this case also implicitly means that there is no successful response within ENDPOINT_TIMEOUT. Because, if there is/are successful responses within this ENDPOINT_TIMEOUT, it would have already been returned by 1), 2) or 3) case.

Empirical Studies

Apparently, most of the examples provided in the last report submission were wrong – because they were considering time property in Exponea server's response as real time. This has been resolved in new implementation. Lets try to replicate Case Studies above by sending requests to our implemented server.

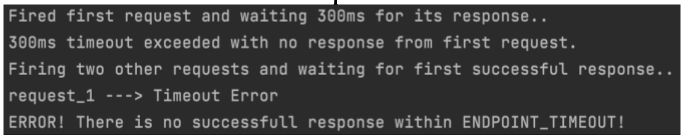
Note that, black screenshots below are taken from terminal after sending the requests. Please, checkout ***with_console_prints branch*** on github repository in order to print results to terminal. They helped a lot for studying & debugging the code, however, I noticed and also read in couple of posts on internet that print() statements to console slows down the server performance. Therefore, I created new branch from ***master branch*** for this purpose, and left master branch intact.

Example №1–№6 are when our server returns ERROR message as below. (i.e., when there is no successful response within ENDPOINT_TIMEOUT)



```
{
    "message": "ERROR"
}
```

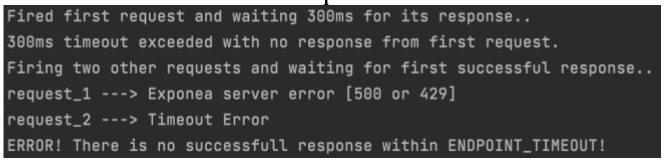
Example №1



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_1 ---> Timeout Error
ERROR! There is no successfull response within ENDPOINT_TIMEOUT!
```

This starts as 3) case, but evolves as 4) case **from the Case Studies above**. while waiting for earliest response from all 3 requests inside as_completed() loop, we got "Timeout Error" response from request_1 which implies that api_smart() execution time exceeded ENDPOINT_TIMEOUT – so, our server returns {'message': 'ERROR'}.

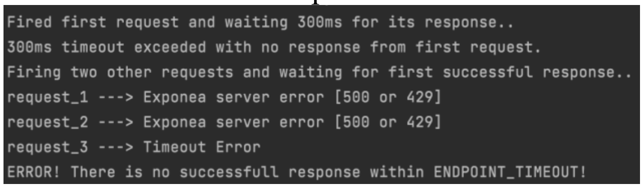
Example №2



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_1 ---> Exponea server error [500 or 429]
request_2 ---> Timeout Error
ERROR! There is no successfull response within ENDPOINT_TIMEOUT!
```

Again, starts as 3) case, but evolves as 4) case, but different from Example №1. This time, while waiting for earliest response from all 3 requests inside as_completed() loop, first earliest response was “Exponea server error” from request_1. Loop continued by awating next earliest response, and got “Timeout Error” response from request_2 which implies that api_smart() execution time exceeded ENDPOINT_TIMEOUT – so, our server returns {‘message’: ‘ERROR’}.

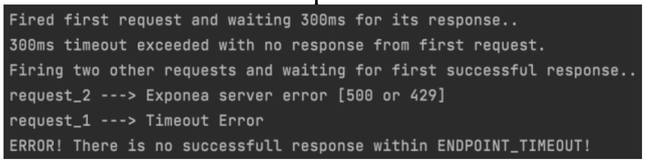
Example №3



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_1 ---> Exponea server error [500 or 429]
request_2 ---> Exponea server error [500 or 429]
request_3 ---> Timeout Error
ERROR! There is no successfull response within ENDPOINT_TIMEOUT!
```

Again, starts as 3) case, but evolves as 4) case, but different from both Example №1 and Example №2. This time, while waiting for earliest response from all 3 requests inside as_completed() loop, first earliest response was “Exponea server error” from request_1, and next earliest response was “Exponea server error” from request_2. Finally, in last iteration of loop, we got “Timeout Error” response from request_3 which implies that api_smart() execution time exceeded ENDPOINT_TIMEOUT – so, our server returns {‘message’: ‘ERROR’}.

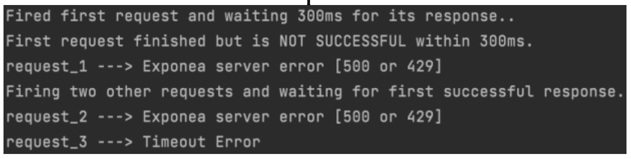
Example №4



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_2 ---> Exponea server error [500 or 429]
request_1 ---> Timeout Error
ERROR! There is no successfull response within ENDPOINT_TIMEOUT!
```

Again, starts as 3) case, but evolves as 4) case, but different from previous examples. This example proves that responses from requests are being collected greedily as they come inside as_completed() loop – so, as shown on screenshot, request_2 was first earliest response, but as it was “Exponea server error”, the loop continued by waiting the next earliest response which is “Timeout Error” from request_1. It implies that api_smart() execution time exceeded ENDPOINT_TIMEOUT – so, our server returns {‘message’: ‘ERROR’}.

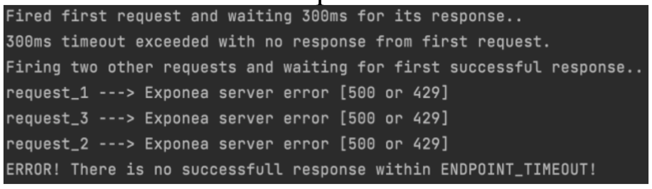
Example №5



```
Fired first request and waiting 300ms for its response..
First request finished but is NOT SUCCESSFUL within 300ms.
request_1 ---> Exponea server error [500 or 429]
Firing two other requests and waiting for first successful response.
request_2 ---> Exponea server error [500 or 429]
request_3 ---> Timeout Error
```

This starts as 2) case, but evolves as 4) case. Its interesting one. As shown on screenshot, first request finished within 300ms but it was not successful because its response was “Exponea server error”. Technically, as soon as we got this reponse from first request, we manually raised TimeoutError in order to jump to “except” clause and fire two other requests. Then while waiting for earliest response inside as_completed() loop, first earliest response was “Exponea server error” from request_1. This is because, although the first request is finished up until now, for the sake of implementation, we still have to **await** it, and as previously explained in Case Studies above – awaiting finished task returns **immediately** – so, the first earliest response had to come from request_1. But still, the loop continued by waiting for the next earliest response because response of request_1 did not have status code 200. Then we get next earliest response as “Exponea server error” from request_2, and loop again continues by waiting for the next earliest response because of same reason – response did not have status code 200. Finally, in last iteration of loop, we got “Timeout Error” response from request_3 which implies that api_smart() execution time exceeded ENDPOINT_TIMEOUT – so, our server returns {‘message’: ‘ERROR’}.

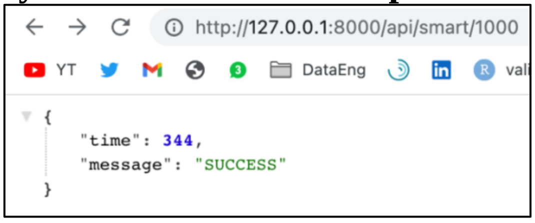
Example №6



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_1 ---> Exponea server error [500 or 429]
request_3 ---> Exponea server error [500 or 429]
request_2 ---> Exponea server error [500 or 429]
ERROR! There is no successful response within ENDPOINT_TIMEOUT!
```

This is 3) case. *NOTE: this is the only example case where the code runs down until the last return statement inside api_smart().* Reason why api_smart() returns ERROR in this example, is different from all previous examples. In previous Examples №1–№5, we returned ERROR because api_smart() execution time exceeded ENDPOINT_TIMEOUT. But in this example, its execution finished within ENDPOINT_TIMEOUT, but all 3 response were "Exponea server error" – so, our server returns {'message': 'ERROR'}. This is rather a rare case, because all requests finished but they finished unsuccessfully, and all these happened within ENDPOINT_TIMEOUT. Note: to catch this case, I set ENDPOINT_TIMEOUT=2000 to allow each request a reasonable time to finish. 1000ms was not enough.

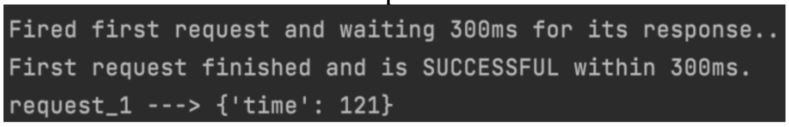
Example №7–№20 when our server returns SUCCESS message along with payload received from Exponea server such as:



http://127.0.0.1:8000/api/smart/1000

```
{
    "time": 344,
    "message": "SUCCESS"
}
```

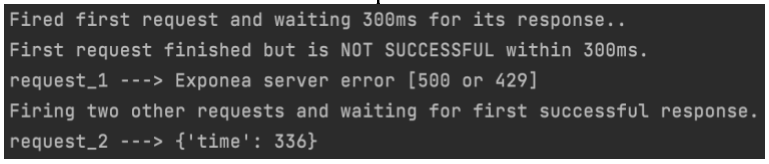
Example №7



```
Fired first request and waiting 300ms for its response..
First request finished and is SUCCESSFUL within 300ms.
request_1 ---> {'time': 121}
```

This is 1) case. First request finished within 300ms and as it was successful (response status 200), we returned it. So, our server returns {'time': 1221, 'message': 'SUCCESS'} json.

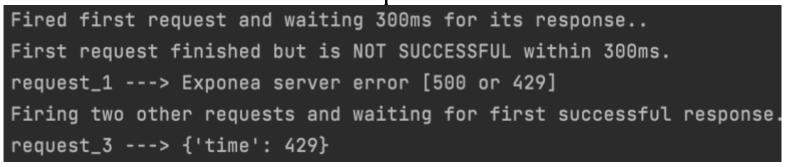
Example №8



```
Fired first request and waiting 300ms for its response..
First request finished but is NOT SUCCESSFUL within 300ms.
request_1 ---> Exponea server error [500 or 429]
Firing two other requests and waiting for first successful response.
request_2 ---> {'time': 336}
```

This is 2) case. First request finished within 300ms but it was not successful because its response was "Exponea server error". As soon as we got this reponse from first request, we manually raised TimeoutError in order to jump to "except" clause and fire two other requests. Then while waiting for earliest response inside as_completed() loop, first earliest response was from request_2. And as it was also a successful response, we returned it. So, our server returns {'time': 336, 'message': 'SUCCESS'} json.

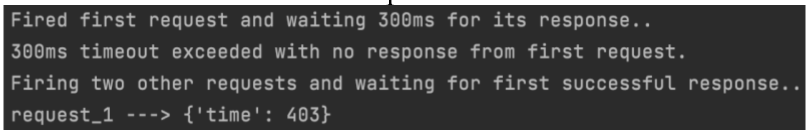
Example №9



```
Fired first request and waiting 300ms for its response..
First request finished but is NOT SUCCESSFUL within 300ms.
request_1 ---> Exponea server error [500 or 429]
Firing two other requests and waiting for first successful response.
request_3 ---> {'time': 429}
```

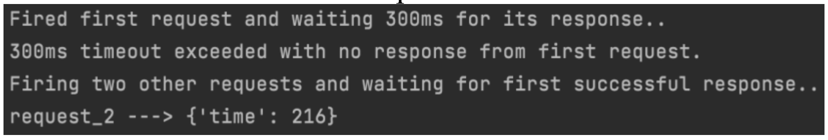
This is 2) case. First request finished within 300ms but it was not successful because its response was "Exponea server error". As soon as we got this reponse from first request, we manually raised TimeoutError in order to jump to "except" clause and fire two other requests. Then while waiting for earliest response inside as_completed() loop, first earliest response was from request_3. And as it was also a successful response, we returned it. So, our server returns {'time': 429, 'message': 'SUCCESS'} json.

Example №10


```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_1 ---> {'time': 403}
```

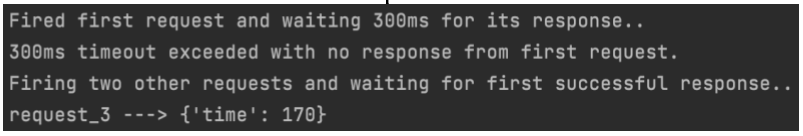
This is 3) case. As said on screenshot above, 300ms timeout exceeded with no response from first request, so, wait_for() threw TimeoutError exception and code continued with "except" clause where it fired two other requests and got response from request_1 as earliest successful response. Note that, response of request_1 is also the earliest response besides being earliest successful response. Our server returns {'time': 403, 'message': 'SUCCESS'} json.

Example №11


```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_2 ---> {'time': 216}
```

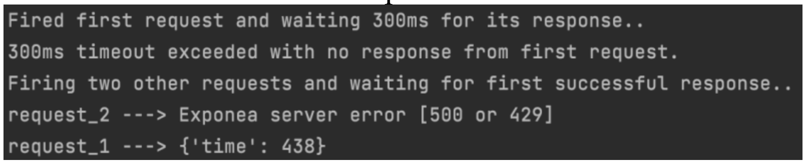
This is also 3) case. Its difference from previous Example №10 is that, this time the earliest successful response comes from request_2. Again, note that, response of request_2 is also the earliest response besides being earliest successful response. Our server returns {'time': 216, 'message': 'SUCCESS'} json.

Example №12


```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_3 ---> {'time': 170}
```

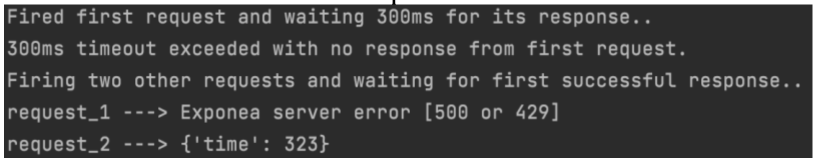
Also 3) case. This time, the earliest successful response among 3 reqests is request_3. Again, note that, response of request_3 is also the earliest response besides being earliest successful response. Our server returns {'time': 170, 'message': 'SUCCESS'} json.

Example №13


```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_2 ---> Exponea server error [500 or 429]
request_1 ---> {'time': 438}
```

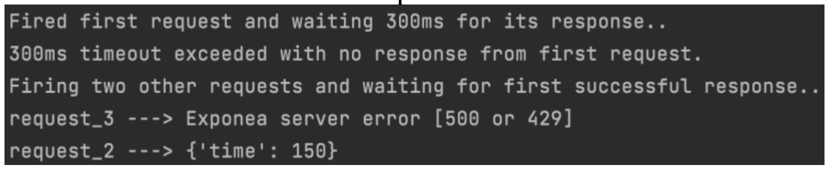
Also 3) case. The earliest successful response comes from request_1. Note that, response of request_2 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Our server returns {'time': 438, 'message': 'SUCCESS'} json.

Example №14


```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_1 ---> Exponea server error [500 or 429]
request_2 ---> {'time': 323}
```

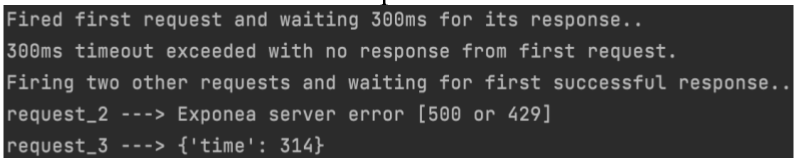
Also 3) case. The earliest successful response comes from request_2. Note that, response of request_1 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Our server returns {'time': 323, 'message': 'SUCCESS'} json.

Example №15


```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_3 ---> Exponea server error [500 or 429]
request_2 ---> {'time': 150}
```

Also 3) case. The earliest successful response comes from request_2. Note that, response of request_3 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Our server returns {‘time’: 150, ‘message’: ‘SUCCESS’} json.

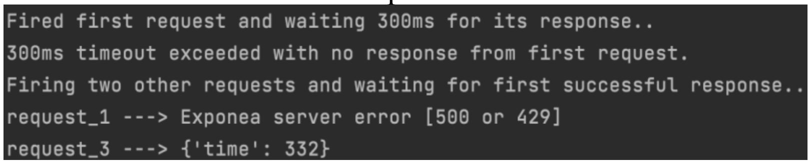
Example №16



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_2 ---> Exponea server error [500 or 429]
request_3 ---> {'time': 314}
```

Also 3) case. The earliest successful response comes from request_3. Note that, response of request_2 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Our server returns {‘time’: 314, ‘message’: ‘SUCCESS’} json.

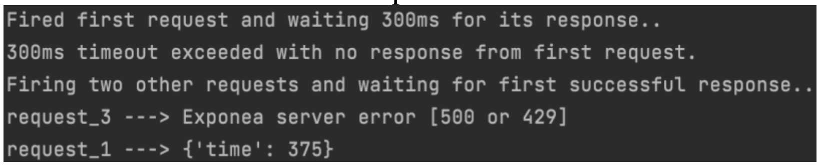
Example №17



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_1 ---> Exponea server error [500 or 429]
request_3 ---> {'time': 332}
```

Also 3) case. The earliest successful response comes from request_3. Note that, response of request_1 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Our server returns {‘time’: 332, ‘message’: ‘SUCCESS’} json.

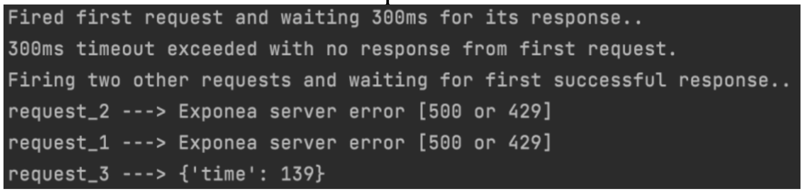
Example №18



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_3 ---> Exponea server error [500 or 429]
request_1 ---> {'time': 375}
```

Also 3) case. The earliest successful response comes from request_1. Note that, response of request_3 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Our server returns {‘time’: 375, ‘message’: ‘SUCCESS’} json.

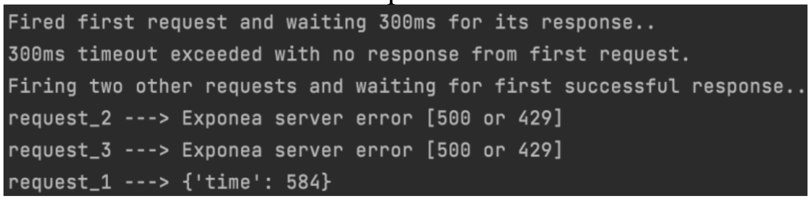
Example №19



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_2 ---> Exponea server error [500 or 429]
request_1 ---> Exponea server error [500 or 429]
request_3 ---> {'time': 139}
```

Also 3) case. The earliest successful response comes from request_3. Note that, response of request_2 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Then response of request_1 was the second earliest response but it was also not succesful, so as_complete() loop continued to the last iteration where it got response of request_3 as successful. Our server returns {‘time’: 139, ‘message’: ‘SUCCESS’} json.

Example №20



```
Fired first request and waiting 300ms for its response..
300ms timeout exceeded with no response from first request.
Firing two other requests and waiting for first successful response..
request_2 ---> Exponea server error [500 or 429]
request_3 ---> Exponea server error [500 or 429]
request_1 ---> {'time': 584}
```

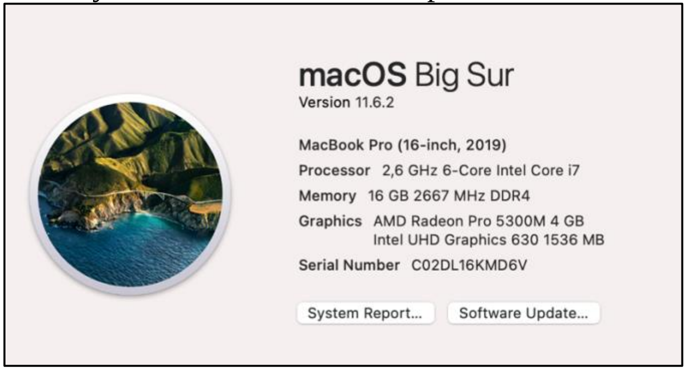
Also 3) case. The earliest successful response comes from request_1. Note that, response of request_2 was the first earliest response but as it was not successful, as_complete() loop continued by waiting the next earliest response. Then response of request_3 was the second earliest response but it was also not succesful, so as_complete() loop continued to the last iteration where it got response of request_1 as successful. Our server returns {'time': 584, 'message': 'SUCCESS'} json.

Performance Benchmark

Actually, finding the proper tool for performance benchmarking took a bit of time. The most mentioned open source tool on discussions was Apache Benchmark, and apparently, it is installed on macOS by default. So, I went with it. Although, it was pretty straightforward and easy to use via terminal, it just enough just for simple tests, and I couldn't predict the performance boundaries of the server in terms of concurrency and number of requests per second. Besides from Apache Benchmark, also tried to use open source tools such as Artillery and wrk, but had some issues during installation, then I finally found out about **Apache Jmeter**, went with it. It is a cool tool. It was a bit hard to understand parameters at first, but it gradually made sense.

Note: I was getting **OSError: too many open files** during initial tests. It was resolved by running **ulimit -n unlimited** command on terminal.

Also, I am running the server on my local machine with the specs as below:



macOS Big Sur
Version 11.6.2

MacBook Pro (16-inch, 2019)
Processor 2,6 GHz 6-Core Intel Core i7
Memory 16 GB 2667 MHz DDR4
Graphics AMD Radeon Pro 5300M 4 GB
Intel UHD Graphics 630 1536 MB
Serial Number C02DL16KMD6V

For performance benchmarking on Apache Jmeter, these will be the parameters that we will play with:

- Number of threads (users)
- Ramp-up period (seconds)
- Loop count (number of iterations per thread/user)

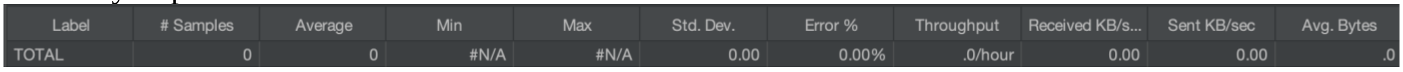
Ramp-up period is an interesting one. According to official user's manual, it tells how long to take to "ramp-up" to the full number of threads(users) chosen. Also, a comment on this stackoverflow post https://bit.ly/jmeter_ramp_up helped better understand its purpose:

"Useful for testing auto-scaling - e.g. for most websites, it's unlikely that you're going to go from 0 to 100 concurrent users instantly. It's more likely that concurrent users will gradually increase (and thereby give the auto-scaling system time to respond). Ramp-up enables this scenario to be tested. – Jon Burgess"

Note that, we will evaluate the tests by checking "**Summary Report**", "**Response Time Graph**" and "**Graph Results**" listeners.

Summary Report summarizes important results of the test on the table. Response Time Graph will help us check response times throughout the test. Finally, we will use Graph Results for checking throughput requests/sec.

Summary Report:



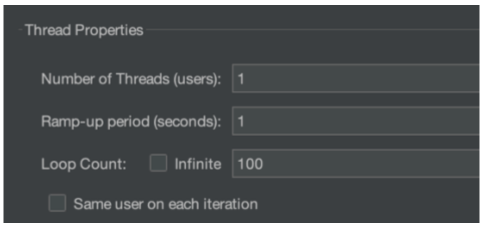
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
TOTAL	0	0	#N/A	#N/A	0.00	0.00%	.0/hour	0.00	0.00	.0

- Samples: number of total requests sent
- Average/Min/Max: is regarding the response time. Response time is the elapsed time from the moment when a given request is sent to the server until the moment when the last bit of information has returned to the client.
- Error%: Percent of requests with errors.
- Throughput: number of requests processed per second/minute/hour by the server. The larger, the better.
- KB/Sec: amount of data downloaded from server during the performance test execution. It is the throughput measured in kilobytes per second.

For all the tests below, we set ENDPOINT_TIMEOUT=1000 ms.

Test №1

Lets start with very simple case where we have 1 user, 100 iteration per user and ramp-up 1 sec. For this and the rest of the tests, we will clear out the box of "Same user on each iteration" – this will allow us to have new user in each iteration. We get the below results on **"Summary Report"** listener.



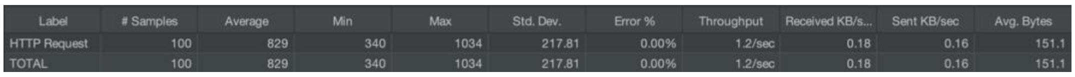
Thread Properties

Number of Threads (users): 1

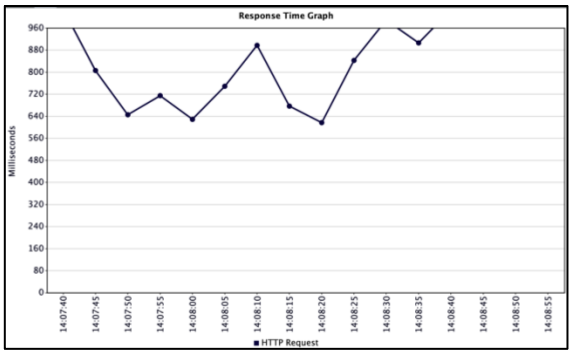
Ramp-up period (seconds): 1

Loop Count: ☐ Infinite 100

☐ Same user on each iteration



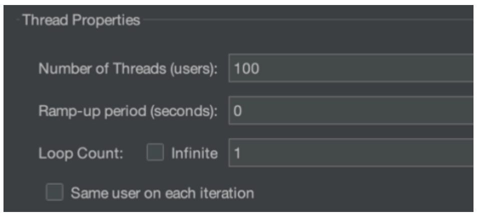
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	100	829	340	1034	217.81	0.00%	1.2/sec	0.18	0.16	151.1
TOTAL	100	829	340	1034	217.81	0.00%	1.2/sec	0.18	0.16	151.1



We see average response time is 829ms. This means that, on average, our server responded within ENDPOINT_TIMEOUT (1000ms). This can also be seen from Response Time Graph above. Also, sometimes we got some responses such as max. response with 1034ms, which is slightly above the time the server is supposed to respond to a single request. The reason for this can be either server's approximate implementation (discussed in case studies) or the transport latency(?). As per https://bit.ly/request_latency, transport latency + processing time = response time, so, this can also be the case. Finally, summary report also indicates that all requests were successful, as we have Error 0.00%.

Test №2

Now, lets try a spike test with 100 users, and ramp-up = 0 seconds. This means that all the threads/users will start instantly once the test will be starts. So, the server will be loaded with total of 100 requests from 100 users within short period of time.



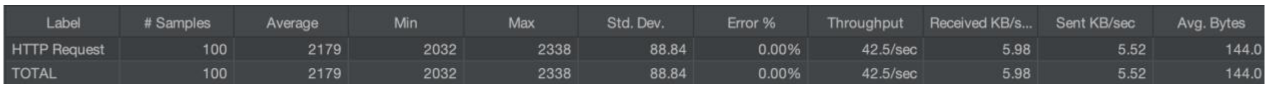
Thread Properties

Number of Threads (users): 100

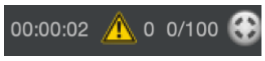
Ramp-up period (seconds): 0

Loop Count: ☐ Infinite 1

☐ Same user on each iteration



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	100	2179	2032	2338	88.84	0.00%	42.5/sec	5.98	5.52	144.0
TOTAL	100	2179	2032	2338	88.84	0.00%	42.5/sec	5.98	5.52	144.0

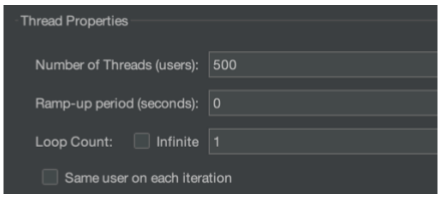


00:00:02 ⚠ 0 0/100

We see that all request finished successfully (Error 0.00%) under the load within ~2 seconds. However, all responses exceeded endpoint’s timeout parameter – we have average response time of ~2179ms.

Test №3

Another spike test, this time with 500 users.



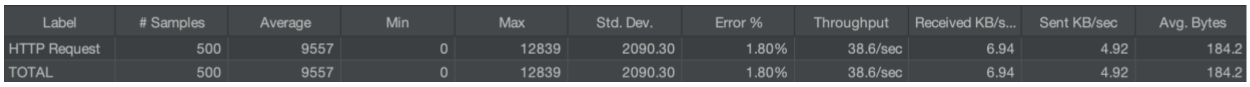
Thread Properties

Number of Threads (users): 500

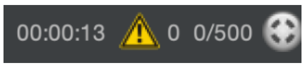
Ramp-up period (seconds): 0

Loop Count: ☐ Infinite 1

☐ Same user on each iteration



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	500	9557	0	12839	2090.30	1.80%	38.6/sec	6.94	4.92	184.2
TOTAL	500	9557	0	12839	2090.30	1.80%	38.6/sec	6.94	4.92	184.2

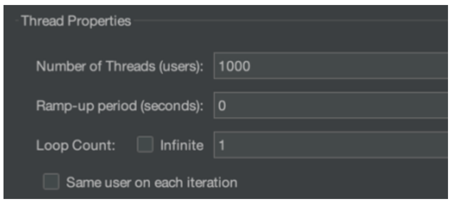


00:00:13 ⚠ 0 0/500

Test finished in 13 seconds, and we had Error rate of 1.80%, average response time ~9557 seconds.

Test №4

Another spike test, with 1000 users.



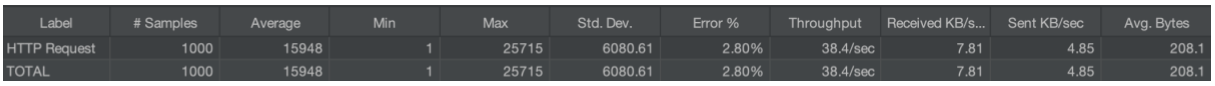
Thread Properties

Number of Threads (users): 1000

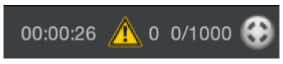
Ramp-up period (seconds): 0

Loop Count: ☐ Infinite 1

☐ Same user on each iteration



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	1000	15948	1	25715	6080.61	2.80%	38.4/sec	7.81	4.85	208.1
TOTAL	1000	15948	1	25715	6080.61	2.80%	38.4/sec	7.81	4.85	208.1

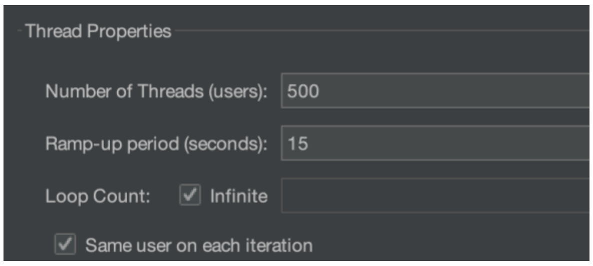


00:00:26 ⚠ 0 0/1000

Test finished in 26 seconds, and we had Error rate of 2.80%, average response time ~15948 seconds. We see that as we increased user size 100 → 500 → 1000 in spike tests, response time and error rate increased.

Test №5

Now, lets continue with load testing for extended period of time. We set 500 users, but with ramp-up=15 seconds. Also, we will allow the test to run 30 minutes. So, we set loop count to infinity, and wait 30 minutes. Finally, we select to have same user on each iteration. This will allow us to analyze how server would behave with 500 users using it 30 minutes, and users will have ramp-up of 15 seconds.



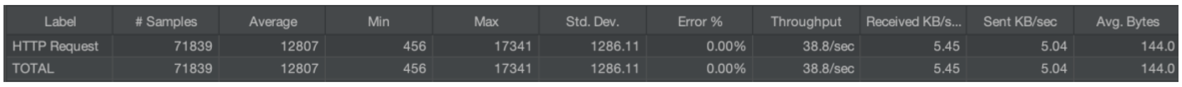
Thread Properties

Number of Threads (users): 500

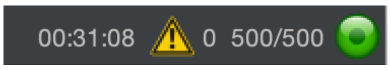
Ramp-up period (seconds): 15

Loop Count: [✓ Infinite]

[✓ Same user on each iteration]



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	71839	12807	456	17341	1286.11	0.00%	38.8/sec	5.45	5.04	144.0
TOTAL	71839	12807	456	17341	1286.11	0.00%	38.8/sec	5.45	5.04	144.0

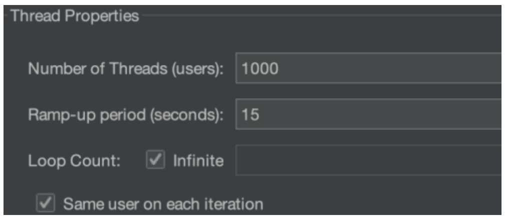


00:31:08 ⚠ 0 500/500 ●

We see that after test run 31 minutes, total of 71839 requests were sent by 500 users to the server, and they were all successfully handled with Error 0.00% and average response time ~12807ms. We can also see that, on average we had **Throughput of ~38 requests/second.**

Test №6

Now, lets increase user size up to 1000, and keep ramp-up same as last test – 15 seconds. We will let the test run for some time to check if Error rate jumps up. If it does, we will increase the ramp-up time for the next test, and see if the server can handle 1000 users in different scenario.



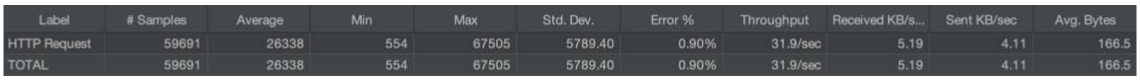
Thread Properties

Number of Threads (users): 1000

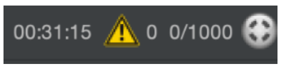
Ramp-up period (seconds): 15

Loop Count: [✓ Infinite]

[✓ Same user on each iteration]



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	59691	26338	554	67505	5789.40	0.90%	31.9/sec	5.19	4.11	166.5
TOTAL	59691	26338	554	67505	5789.40	0.90%	31.9/sec	5.19	4.11	166.5



00:31:15 ⚠ 0 0/1000

We see that after test run 31 min, we had total of 59,691 requests made by 1000 users. This happened with 15 seconds ramp-up. Also, server couldnt handle some of requests during test – as results show Error 0.90%. However, this time, throughput decreased down to ~32 requests/sec.

We can conclude that, with my local MacOS machine, our server had average Throughput of 35-40 requests/sec.

Along the way

You may notice that the script has got a different shape compared to first submission. Main reason is that there were lots of different obstacles that emerged every time I tried to add a new piece of code, so I had to adjust along the way to address those exceptions/errors. Few points along the way:

- As mentioned at the beginning of report, asyncio.gather() is actually working different than I was thinking. So, in order to implement the assignment correctly, I had to look for other options - asyncio.wait_for(), asyncio.shield(), asyncio.as_completed() helped to address this.
- At one point, I had to switch from httpx to aiohttp**,** because httpx were throwing *RuntimeError: The connection pool was closed while 1 HTTP requests/responses were still in-flight* that I wasn't able to resolve. You may refer to recent discussion around the error on https://bit.ly/httpx_runtime_error.
- At one point after switching to aiohttp, I was getting *aiohttp.client_exceptions.ClientConnectorCertificateError*. Installing certifi and using its certificates addressed the issue on https://bit.ly/aiohttp_certifi.
- Then I was getting *aiohttp.client_exceptions.ContentTypeError*, this was addressed by https://bit.ly/aiohttp_content_type. I set response.json(content_type = None) and introduced exception handling for ContentTypeError.
- Also had to put "except" clause in send_request() function to handle other possible exceptions such as when task gets cancelled, or in case of clients connection error, or client OS error.

Final notes

Big thanks to the team for this interesting assignment. It was a fun ride. I absolutely loved every step of it, be it code development/engineering/case studying or performance benchmarking of the server. I got to say, I was happy seeing the performance during the benchmarking, considering it's my first backend server building ☺ I'd much appreciate hearing your feedback on pros/cons of it.