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>

**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:

* To your third point on feedback:
  + *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 task results. As a result, the previous code was waiting for all the tasks to be finished before sending the first successful response. But, as we want to process tasks 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 from awaitable tasks.

*🡪* 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.

* To your second point on feedback:
  + “*Second and third requests are sent after the first request finishes instead of waiting 300ms since the request has been started.”*

*🡪* thecause, 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.

**Case studies**

First, let’s explore the pseudo-code of our new approach:

Text

Description automatically generated with low confidence

Graphical user interface, text, application, email

Description automatically generated

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

* by client session if it exceeds ENDPOINT\_TIMEOUT
* by async.wait\_for() if it exceeds 300ms
* by manually raising

The reason for this is that, on the other side, send\_request() can handle all exceptions that may occur during the requests and returns the respective response status back to api\_smart(). This helps api\_smart() to evaluate the requests according to their response statuses, and not deal with exceptions other than TimeoutError. Let’s see how the implementation works below.

Once the API gets a request through endpoint, api\_smart() starts by firing the first request and waiting for its response 300ms:

1. If within 300ms, first request successfully finishes (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, and code goes to “except” clause. Idea is that, as our first request failed, it is time to fire two other requests **without** waiting 300ms to finish.
3. If 300ms exceeds, 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, thus, 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 killed by wait\_for() once the timeout occurs. Reason is that, even if the timeout occurs, we still need the response from first request (see below).
4. As said above, if 2) or 3) case happens, we continue with “except” clause for handling the thrown TimeoutError exception. This is where we fire two other requests and process all requests greedily as they finish using asyncio.as\_completed().

Technical Note: as per code, inside asyncio.as\_completed() loop, we get earliest next response by awaiting tasks of 3 requests. Remember that, as of now, first request is either finished unsuccessfully (2) case) or still did not finish (3) case). So, during the loop, awaiting the first request can happen as either of below:

1) First request is finished and we will await its finished task – awaiting the finished task returns its result immediately (<https://bit.ly/await_finished_task>)

2) First request is **not** finished and we will await its task.

As we greedily loop through the finished tasks, we 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.

1. Last case is when during any of above cases, total time spent by implementation exceeds ENDPOINT\_TIMEOUT. This is approximately addressed by allocating timeout for each request in a way that when total time spent by endpoint exceeds ENDPOINT\_TIMEOUT, it returns ERROR. It also implicitly means that there is no successful response within ENDPOINT\_TIMEOUT. Because, if there is/are successful responses within this timeout, it would have been already returned by 1) or 4) cases.

**Examples**

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 with ENDPOINT\_TIMEOUT=1000:

**Deepnote: our uvicorn server always respond successfully within given timeout with status code 200.**

**https://webdock.io/en/docs/webdock-control-panel/optimizing-performance/how-benchmark-your-server-apachebench**

**Graphical user interface, text, application, email

Description automatically generated**

**https://stackoverflow.com/questions/19527097/does-print-statement-will-make-performance-issue-for-a-website**

Edge cases when 300 > endpoint timeout parameter

Endpoint timeout exception can or can not kill task\_1? Does shield protect task\_1 from all timeout exceptions? Probably can kill. Because once the endpoint timeout exception occurs, the whole session closes, and there is no way that task\_1 is alive.

**Along the way**

You may notice that the script has got a different shape after the update. The reason is that, there were lots of different errors that emerged every time I tried to add a new piece of code. And for addressing these exceptions/errors, I had to adjust along the way. Overall, here are few points that kind of describe the evolution of script.

* I came up with one stackoverflow post that also involved multiple requests, and one of answers advised to have a single client session for all requests, rather than having separate ones. I grabbed the idea as a baseline – to have a single session for all requests. This would also help to properly implement the task in terms of ENDPOINT\_TIMEOUT, as we will have a single session.
* As mentioned at the beginning of report, asyncio.gather() is actually working different than I was thinking. So, in order to implement the task correctly, I had to look for other options - asyncio.wait\_for(), asyncio.shield(), asyncio.as\_completed()
* 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.

As far as I understand, the server is working as expected and the endpoint always returns a response within the given timeout. Also, the responses are presumably (?) correct – this needs discussion with the team. However, there are few exceptions that I still couldn’t handle:

1. ometimes, we get aiohttp.client\_exceptions.ClientOSError: [Errno 1] [SSL: KRB5\_S\_INIT] application data after close notify (\_ssl.c:2683). According to <https://bugs.python.org/issue39951> and <https://bit.ly/ssl_ignore>, it is an error that can be ignored when closing SSL connections. Bug is recently fixed for Python 3.11 and is waiting for 3.10 and 3.9. So, possible solution now would be to upgrade my current python 3.8 to 3.11. Due to time constraints, also as upgrade itself can produce other errors for the code, I decided to skip it.

**Discussion**

In this report we will discuss the workflow of api with various use cases. Let’s start:

First of all, we have only one api endpoint, which is /api/smart/<timeout\_parameter> which performs up to 3 HTTP requests to Exponea Testing HTTP Server (<https://exponea-engineering-assignment.appspot.com/api/work>) and returns first successful response. Note: if no successful request returned within the given timeout, api responds with error messages for each of 3 requests that failed.

Algorithm is built in this way:

First it sends one request to the testing server. If there is a successful response within 300 milliseconds, then the endpoint returns this response and doesn’t fire other requests.

But if the time of first request is bigger than 300 milliseconds, it fires concurrently (asynchronously) two other requests, and then returns the first successful request from any of the 3 requests, including the first one.

Example №1:

İf first request to testing server returns time equal to 270, then api returns answer like this {“time”: 270}

Example №2:

İf first request to testing server returns time equal to 350, then immediately it fires another 2 requests, which are 280 and 528 respectively, and returns the fastest one, means {“time”: 280}

Concurrent requests are made with help of httpx and asyncio, one of which helps to send the requests asynchronously, other one to gather data from the responses of requests.

But there are other cases, when there appear errors, such as Server timeout, Connection timeout, Internal server error (which comes from testing server), which require to be handled. In this case, if one request returns an exception, it just does nothing but writes it down and deals with other requests. For example, if second request returned an error, it just skips it and returns first successful response from first and third request, and so on. But in case if all requests are failed, it returns json response with the error message for each of the request.

Let me show some use cases:

Example №3:

First request to test server returned 350, then second and third are fired, second returns an error, and third returns 420. The response time will be 350

Example №4:

First request to test server returned 400, then second and third are fired, second returns an error, and third also returns an error. The response time will be 400

Example №5:

First request to test server returned an error, then second and third are fired, second returns 340, and third returns 450. The response time will be 340

Example №6:

First request to test server returned a server error, then second and third are fired, second returns a timeout error, and third also returns a timeout error. The response will be like that {1: Server error, 2: Timeout error, 3: Timeout error}

During the testing period, the testing server behaved differently. Sometimes it worked perfectly without errors, sometimes it returned almost all errors each time. It is advised to test Api regularly to avoid misunderstandings in app workflow.

1. ~~If client session exceeds ENDPOINT\_TIMEOUT, it throws TimeoutError exception and the code will go to “except” clause. In the meantime, if there are any unfinished requests, each of them gets killed. So, as all the requests are finished by now, and as each of them has response status other than 200, asyncio.as\_completed()~~~~loop will finish without returning, and we will continue with next line which returns ERROR~~**~~.~~** ~~Note that, we stated that~~ *~~all requests are finished by now and each of them has response status other than 200~~* ~~– the reason why this holds true is that if there was any successful response before ENPOINT\_TIMEOUT exception, it would have been returned by algorithm by either 1) or 4) case.~~