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 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:

* The previous code was creating a new client for each request inside get\_time() coroutine function. Better and more accurate option will be to tackle all requests within single client. This way, we can assign endpoint’s timeout parameter to client’s timeout parameter.
* 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.

* 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.

**Code discussion**

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

Graphical user interface, text, application

Description automatically generated

As seen above, we have nested try-except clauses, and both of them handle TimeoutError exception. We will below explain in which case, which TimeoutError exception will be handled by which block. Essentially, during execution of code, there are 3 cases in which TimeoutError can be thrown:

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

If client session exceeds ENDPOINT\_TIMEOUT, session throws TimeoutError exception and this will be handled by outer “except TimeoutError:” block and it will return ERROR. This means that session exceeded endpoint timeout, and we return ERROR as we have no successful response within endpoint timeout. However, if there was any successful response within endpoint timeout, or if all the requests were unsuccessful within endpoint timeout, these would have already happened within inner try-except block, and the inner block would have returned earliest successful response or ERROR, respectively (more on this below).

Now lets explore how inner try-except block works. It starts with firing first request and waiting for its response 300ms:

1. If within 300ms, first request successfully finish, then we return first request’s response.
2. If within 300ms, first request finishes, but it is not successful, then we manually raise TimeoutError, and handle the exception in inner “except TimeoutError” block. Idea is that, as our first request failed, it is time to fire other two requests without wasting time.
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, it is time to fire other two requests. Note that in this case, we are sure that first request did not finish within 300ms, because if first request finished within 300ms, 1) and 2) cases above would have already been run and returned accordingly. 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).

Technical Note: the reason why we manually raise TimeoutError on 2) case is that in both 2) and 3) cases, we have to proceed with firing two other requests and considering that firing two other requests occurs during inner “except TimeoutError:” block, we have to manually raise TimeoutError in order to jump to that block.

1. If 2) or 3) case happens, we continue with inner “except TimeoutError:” block for handling the 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 result by awaiting tasks. Remember that, as of now, first request is either finished unsuccessfully (2. case) or still did not finish (3. case). So, the loop can be in two cases:

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

1. first request is not finished and we will await its task.

As we greedily loop through 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.

Text

Description automatically generated

**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.