File Chunking Server

How to run:

In the source code, you will see a nginx config file that is used for the nginx load balancer. That of course, requires the installation of nginx.

So the aiohttp server is called by running the file server.py in the Aio server directory:

/Assignment2/File_Chunking_server/src/Aio_server$ python3 server.py

The server will then begin to run on localhost:8080.

You will see an index page where you can upload a file and specify the chunk. The file will then automatically be downloaded.

Sian

Architecture

The application was mainly written in Python with a aiohttp frontend local server, nginx load balancer and three django backends hosted on AWS.

We wanted to build an application that would be able to handle incoming requests at the frontend and distribute the actual ‘work’ of chunking the file across different versions of the same backend.

Aiohttp was used as a frontend application as it allowed for asynchronous functions and good support for streaming content through HTML requests. Both of which I used heavily within the application. The async allows for asynchronous requests i.e the program is not waiting for a reply from the backend when sending a request to do more work. The streaming ensures that the files do not write to the server memory and overload it.

I then implemented a nginx load balancer in the system that distributes the POST requests made from the AIO server to the three django servers hosted on aws. The load balancer used a method ‘least_conn’ which means it would go to the server with

the least amount of connections ongoing at the time of request. This was to improve the speed of the requests and boost performance in general.

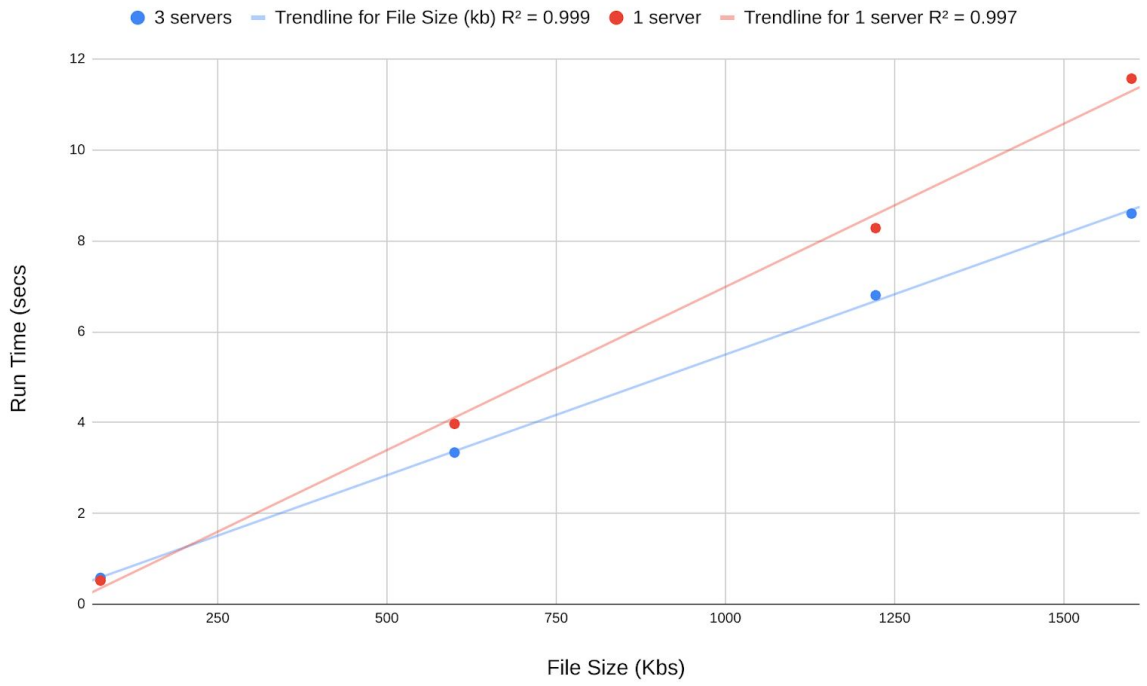
Lastly there were three django backends that all did the same thing in the backend. This created a distributed system which increased reliability(failure handling), performance and scalability.

Of course, there are many flaws in the system and I recognise this. Aiohttp is a very good alternative to django/flask in terms of concurrency however it is not a very well known library and I did have an issue with lack of resources and documentation. I did not get the queues you specified in the spec working, honestly I am not sure if this is entirely possible in Aiohttp.

Testing

I uploaded files of size 77KB, 600KB & 1.6 MB of randomly selected text and submitted them to chunk for all letters(a-z). I did this using just one backend server and then using the 3 servers and load balancer.

The redline shows the results for one backend server in the system and the blue shows the results for all three routing from the load balancer.



Nereya

It is clear to see that the blue line shows an improvement in the speed of the program and it looks like this is greatly improving the larger the file is.

It is also good to see that on both lines that the size of the file does not impact the performance with relatively straight lines indicating the system does not dissimprove when reading in and out larger files.

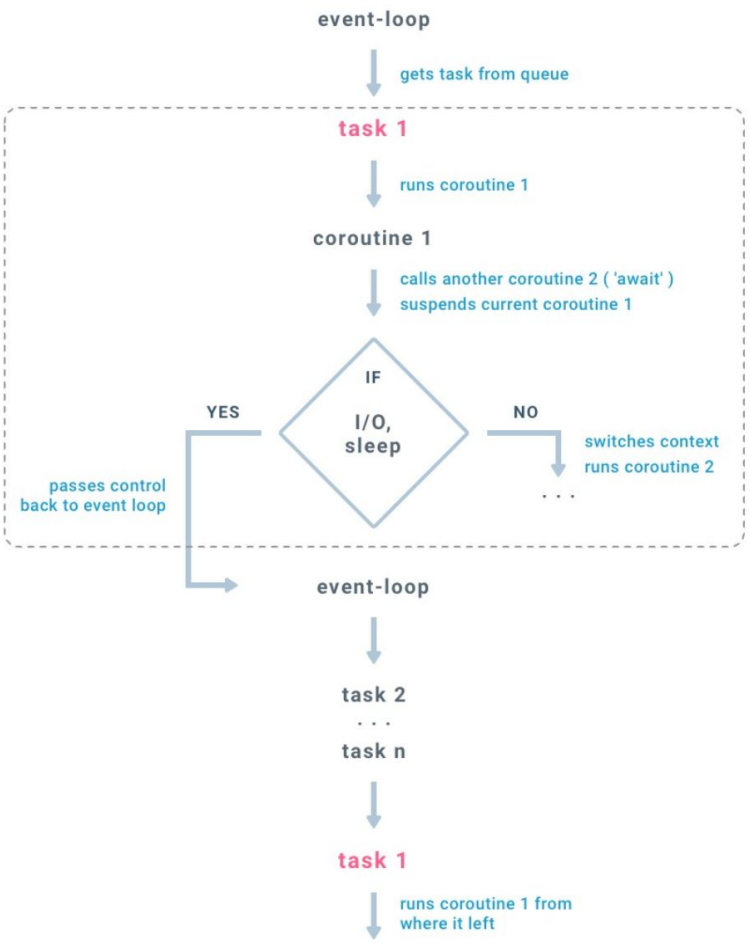
Ideally, it would be great to see how the speed of larger and smaller files performs with many clients on the server for both tests run above. I would like to think that the difference in using one server and three in the background would be even greater.

To combat the synchronous behaviour of python, we used the Python asyncio library along with the aiohttp package. This allowed us to write concurrent code which in turn allowed us to write asynchronous clients and servers. When the application starts running an event loop is kicked off , this is the core of every asyncio application. The event loop plays a vital role as it is responsible for:

- registering all the tasks to be executed,
- executes the tasks,
- delays tasks,
- cancels tasks,
- handle the different events related to the operations.

We took advantage of the above mentioned Python features to test and illustrate the system's ability to handle multiple client requests concurrently. This was achieved by creating Client duplicates that made requests (POST/GET) to the running server. These requests were run at random time intervals quite close together which increased the likelihood that there would be multiple requests hitting the server simultaneously. A distributed approach was used on the server side to handle the multiple client requests, when the first hit server was too busy to handle the incoming request, the request was rerouted to the next available server. The “await” statement in the client code is activated as the client waits for a response from the server other parts of the code can run such as allowing for the previous/next Client with an established client connection to the server to send data(file) to the server. When an “await '' statement is completed, e.g. the server responds to a client a callback occurs and the code after the await is completed. The “await” statements allow for interrupts in the code so that there isn’t a stand still while responses are being sent and waited for. This allows for more asynchronous code. The below image illustrates how the strategies discussed above take form in our code.

Thread


event-loop
gets task from queue
task 1
runs coroutine 1
coroutine 1
calls another coroutine 2 ('await')
suspends current coroutine 1
IF
I/O,
sleep
YES
NO
switches context
runs coroutine 2
. . .
passes control
back to event loop
event-loop
task 2
. . .
task n
task 1
runs coroutine 1 from
where it left