# Project: Proxy herd with asyncio

<FIRSTNAME LASTNAME>, *University of California – Los Angeles*

## 1. Abstract

This project asked for a professional opinion on whether Python's *asyncio* library is suitable for the architecture of an "application server herd". This architecture requires the ability for fast inter-server communication. It is the opinion of this engineer that *asyncio* not be utilized to implement said architecture. The library provides great flexibility and ease of use, but the recommendation of this engineer comes with the caveat that we require multiple cores and a large memory system.

## 2. Introduction

Often in computing, the majority of time spent can be spent waiting on I/O operations. For that reason, it can speed things up drastically to take advantage of this waiting time to perform other operations. This approach is called *asynchronous*, as opposed to the traditional synchronous approach.

Python's attempt at the implementation of asynchronous I/O is via the *async* library. This provides multiple levels of both primitives and higher-level functions which allow programmers to manipulate the behavior of the process in various ways. These approaches will be mentioned subsequently, after the more general aspects of asynchronous I/O have been laid out.

We will now discuss the lower-level primitives and types provided by the async library, which build the framework for the more convenient high-level API used in the project.

### 2.1. Low-Level Terminology

I/O in this system is built upon the idea of a *socket*. This is an object that mimics the behavior of a file descriptor, allowing a program to both read from and write to it. These sockets are abstracted upon by the library to form a *transport*. This allows the program to communicate with another process through a TCP protocol, a pipe, or another method entirely without a change in API. Because the async API functions on the surface in much the same way as the synchronous API, this allows the transition of synchronous to asynchronous code to often entail little more than a change in API, provided that the given function has been implemented asynchronously.[1]

To communicate asynchronously with a transport, Python takes advantage of the idea of a *callback*. A callback is a function called by a transport upon the occurrence of a given event. In asynchronous I/O, this often means the completion of a read or write, or it means a new connection has been established. This functions much like a continuation, returning to the context of the original call to read or write and allowing the process to resume with its local memory intact. [2]

The low-level process API abstracts a procedure into the idea of a *subprocess*. A subprocess is a process connected to the main program through pipes. By utilizing asynchronous subprocess, we can allow subprocesses to await a response whilst performing other tasks. These are wrapped by a *process* object, which allows for simple communication and observation of subprocesses.[3]

Asynchronous I/O of any type centers around an *event loop*. This loop runs in the background of many event-based systems, checking for any event which should correspond to a given behavior. In the example of asynchronous I/O, this often is in the form of a read or write, for example; completion of the I/O operation is the "event" which triggers the waiting process to continue its operation. The event loop does this with a combination of *events*, *conditions*, and *semaphores* which are not particularly important to a discussion of implementation; we can abstract these to the level of a *queue* of operations waiting to be performed.[4]

### 2.2. High-Level Terminology

Having discussed the framework that the relevant API is built upon, we can now begin our discussion of the actual framework of the *asyncio* library.

This discussion begins with a set of classes known in Python as *awaitables*. These are processes which can be run asynchronously and thus awaited upon by the relevant process. Whilst waiting on completion, the event loop can perform other tasks, adding the awaitable to the event queue when the condition, often completion of input or output over a network, is complete. These are broken farther into three main categories.[5]

Python defines a *coroutine* as any awaitable which can be awaited by another coroutine. These are defined via

[1] Asyncio

[2] Asyncio

[3] Ibid.

[4] Ibid.

[5] Ibid.

functions which include the *async* keyword prior to their definition. Calling them returns a coroutine object which can be run; the actual asynchronous operation will not be run until the function is called prepended with the keyword *await*. This causes the system to begin the coroutine, only resuming execution once this coroutine has been completed.[6]

The second type of coroutine is not generally used directly in code built upon the *asyncio* API; these are called *Futures* and represent the eventual result of the execution of a coroutine. These allow us to plan for the execution of various callbacks, effectively amounting to a style of code not dissimilar to the continuation-based programming of scheme.[7]

Coroutines can also be wrapped into a *Task*, which automatically schedules them to be executed by the event loop. These coroutines are not immediately awaited upon, and so can be used to schedule parallel asynchronous operations. The task object can be used to cancel the coroutine or to await it; it thus allows for more flexibility than a traditional coroutine would. These function as a generator, as opposed to a Future, which runs once and returns a result. With a coroutine, we can run a varying number of times, pausing at an await statement each time.[8]

### 2.3. Program-Level API

Writing programs in Python that exploit asynchronous I/O is extremely simple. The *asyncio* library relies on a few fundamental functions and keywords that can allow nearly direct translation from synchronous to asynchronous code.

The first and primary keyword in *asyncio*, as previously mentioned, is the *await* keyword. When placed directly before a function call, this causes the process to yield control to the event loop, allowing for another subprocess to run. Because this requires only the addition of a single keyword, very little change is actually required from the standpoint of program structure. This only requires that the subsequent function be asynchronous.

If we wish to create a *Task* and not immediately wait for the response, we use the function *asyncio.create_task()*. This function takes a coroutine as an argument, and schedules it for execution in the event loop. It returns a *Task*, allowing us to easily connect with the coroutine.

If we wish to schedule multiple tasks concurrently, we can use the function *asyncio.gather()*. This function schedules all tasks and coroutines passed to it, and returns a list containing the return values of each. We can even delay the propagation of exceptions, instead placing those within the return list and dealing with them once all are done executing. We can also deal with exceptions as they come, if we prefer, or even end execution of all coroutines entirely.

To run an asynchronous program, we use the *asyncio.run()* function. Asynchronous routines can only be completely run via the *await* keyword, but the program in itself is often synchronous; to run an entire asynchronous program we pass it as the argument to this function. In practice, this changes very little, since we can simply change the call to *main()* to a call to *asyncio.run(main())*.

Our last API to discuss regards asynchronous server communication. The call we use to establish a server is *asyncio.start_server()*. We pass this an asynchronous function that will be called whenever another process establishes communication with our server.

The abstraction that *asyncio* provides for communication is that of a *Stream*. Upon a call to *asyncio.open_connection()*, we are returned a *(StreamReader, StreamWriter)* pair, which function exactly as one would imagine. These implement their own asynchronous read and write member functions which must be called with the *await* keyword.

## 2. Language-Level Discussion

Python is by its nature synchronous. It is used to running lines of code one after another in a single thread, waiting to execute the next line of code until the previous has finished. For this reason, writing asynchronous code requires extra caution. While it is easy to translate synchronous code into asynchronous, it is also easy to leave a function in its synchronous form, which can cause all asynchronous operations to be halted while the program waits on a single synchronous operation. This can cause an entire asynchronous program to run at the speed of a synchronous one.

There is another issue with the attempt to translate synchronous code into its asynchronous equivalent; Python asynchronicity is still in development. While the accepted methodology for asynchronous I/O in particular has now been standardized, asynchronous versions of other libraries are often still in development. Were our framework to depend on a Python library which did not yet have an asynchronous form, we would again fall down to possibly the speed of a synchronous program, which is clearly completely unacceptable.

The relative cleanliness and ease-of-use of Python comes with a cost; the high-level nature of Python can lead to various inefficiencies that may become too much of a burden for use in our system. This cost comes in two major forms: memory and performance.

### 2.1. Memory

[6] Asyncio
[7] Ibid.
[8] Ibid.

Python operates at a high level. It hides the underlying pointers from the programmer, and instead allocates all objects dynamically, leaving deallocation to be done by a garbage collector. For this reason, it can often be space-intensive to handle large amounts of data in Python. As our framework will be expected to handle ephemeral video data, I believe that this could possibly result in massive space costs, especially since even the minor prototype developed for this project involved 5 copies of each piece of data stored.

Additionally, *asyncio* has a history of memory leaks. As programs have been known to run thousands of coroutines, this can lead to serious problems. Additionally, asynchronous code is difficult to run from the Python interpreter, so debugging can be especially difficult.

In addition to memory leaks, programs based on *asyncio* tend to be generally memory intensive. Each subroutine called within a multiprocessing system is given its own heap, hence the potentially hundreds of subroutines can result in a massive overhead memory cost.

### 2.2. Performance

Performance can also be a major issue when it comes to Python applications. It has a bit of a reputation for being inefficient. In my estimation, a large portion of this is due to the dynamic typing used by the language. Python allows an extreme amount of flexibility; variables, functions, and all kinds of objects can be assigned and reassigned to another value of a different type. While this is of great benefit from the perspective of simplicity and easiness of development, it tends to harm in the way of optimization.

Without clearly defined representations of variables, Python cannot effectively be compiled fully into a single bytecode, since it must be prepared for the contingency of any variable type. This is not dissimilar to a C dynamic library which cannot be optimized, since we do not have access to the function call that will be done at runtime. This means that Python also has to do constant type checking so as to throw an exception if the operation is not allowed.

## 4. Conclusion/Recommendations

Due to the massive and often unavoidable memory and performance costs of a Python-based system, it is thus my recommendation that unless we can manage very powerful hardware, we stay away from Python. It is also worth mentioning that there is still much legacy code that was developed for Python2 and will not run in Python3. While this may become a very viable option in the future, as the asynchronous and Python3 libraries develop, there are better options for our system in the current day.

I would recommend strongly that the team look into development using Node.js. Node.js is a runtime built upon JavaScript which is built upon the idea of multiprocessing. The system exposes the concept of the event loop, in fact utilizing it as the fundamental element of any Node.js program.Node.js is at its heart event driven. For these reasons it is one of the best methods for developing asynchronous programs.[9]

Rather than coroutines, Node.js utilizes an Object called a *Promise*. These are objects which represent future function calls. We can set the behavior of the program based on the result of the routine. Thus, a promise works much like a continuation; we can invoke an asynchronous call with contingency plans for both success or failure and continue to perform other operations while this asynchronous call is acting. This is very similar to passing a continuation to a function in scheme – even more so than Futures, since a Future represents the result whereas a Promise allows us to set behavior based on the performance of an asynchronous call, effectively restoring the environment of before the call to allow operations to continue.[10]

Node.js is also higher performance than Python. Since Node.js is written in a combination of C, C++, and JavaScript, we can reasonably assume that it is a lower level language, and thus allows more performance optimizations than Python. Additionally, it is a runtime environment for JavaScript, which is a language run by a Just-In-Time compiler, allowing hotspots of strongly typed code to be compiled to bytecode and executed with even greater efficiency than that of traditional higher-level language code.[11]

To conclude, it is the belief of this engineer that Python's *asyncio* library is likely too time and space inefficient to be utilized in a large server like ours. With speed and memory being paramount to the performance of our servers, I believe we need to prioritize these elements, and the additional costs of learning and organizing JavaScript programs is likely well worth it in the long run. We may want to keep our eyes open, however, as Python's asynchronous capabilities have not yet reached their apex.

## References

[1] "Asyncio - Asynchronous I/O." Asyncio - Asynchronous I/O - Python 3.8.3 Documentation, docs.python.org/3/library/asyncio.html.

[2] Node.js. "Guides." Node.js, nodejs.org/en/docs/guides/.

[9] Node.js
[10] Node.js
[11] Ibid.