Asyncio Research Report

By Rishab Abdulvahid

Abstract

This report is designed to answer and discuss three critical questions regarding asyncio:

A) What is asyncio and how does its design lend itself to serve as an effective API for asynchronous operations?

B) How can asyncio be leveraged to design a system that receives communications, and must communicate with itself, in ways that are dynamic?

C) How do the fundamental language decisions taken to create python affect the implementation and functionality of asyncio as an asynchronous API, in comparison to something like Node.js in JavaScript?

While doing so, a specific prototype project will be discussed: a server script that is designed to support bidirectional communications amongst five different servers which receive various client requests that they must propagate amongst themselves. This project was built using the asyncio module and will serve as a strong through line for an analysis of asyncio.

1. An Introduction to Asyncio

Asyncio is a module in python (added to version 3.4 of the language) designed in order to provide asynchronous I/O functionality for python programmers. Asyncio accomplishes this by using a combination of coroutines and an event loop, which allows a single-threaded application to emulate (and, often, surpass) performance of a parallelized implementation in an I/O framework: these terms will be discussed in greater detail later in the report. To begin with, it is important to note the difference between concurrency and parallelism, and state how asynchronous programming fits into these programming paradigms.

1.1. Concurrency, parallelism, and asynchronous I/O

Concurrency, in practice, is a principle in programming by which multiple operations are given the freedom to run alongside one another (concurrently, as the name implies). However, the methods by these simultaneous operations are managed varies amongst programming paradigms. *Parallelism* is a paradigm that achieves concurrent execution by running multiple elements of a program at the same time on multiple separate compute units. For example, if a for-loop is designed to increment a variable 1000 times, 4 cores could do this in parallel by incrementing that variable 250 times each. *Multithreading* is another programming paradigm that aims to achieve concurrency by allowing one process to run amongst multiple units of execution (threads) within it. Due to design reasons, Python is inherently a single-threaded programming language. However, this is where asynchronous programming and asyncio come into play. *Asynchronous operation* is a programming paradigm by which concurrency is achieved by allowing units of execution to interrupt one another, even though only one thread may be running at one time. While a multithreaded process operates under the granularity of multiple threads under a process, an asynchronous program can operate with the granularity of multiple "routines" under a single thread. Thus, asynchronous programs respond well to requests that are of a dynamic nature (such as servicing I/O requests) and emulates the functionality of other concurrency paradigms, such as multithreading and parallelized operation.

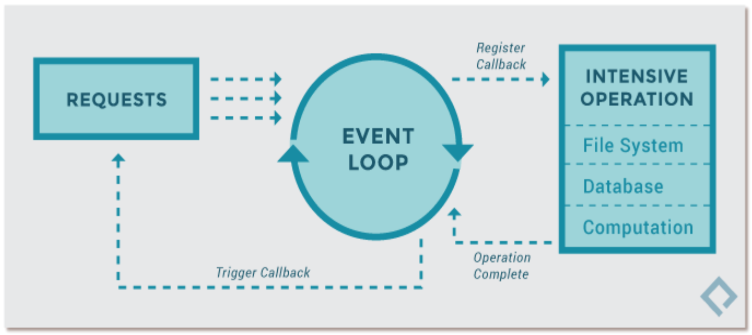
1.2. Coroutines

A natural question following from this point is how Python is able to achieve asynchronous execution even though the language is, inherently, single-threaded. *Coroutines* are functions built within asyncio that can be interrupted by other coroutines. This type of concurrent execution is managed by a lower-level structure known as the event-loop: this structure decides which coroutines are to be currently executed and can suspend execution of one coroutine for another. At any time, python is running coroutines on a single thread, but the asyncio module allows the process to effectively give the illusion that it is simultaneously running multiple routines on multiple processing units, due to the way the event-loop can dynamically schedule and de-schedule routines. In this manner, asyncio can provide asynchronous execution by better leveraging the processing power of your CPU: even though only one thread is running at a time, your CPU is no longer spending down time by yielding and blocking. Thus, asyncio can effectively grant asynchronous functionality to python while still allowing programmers to create modularized code.

2. The Event Loop

Given that the event loop is the crux of the functionality of asyncio, it's worth taking a more detailed look at how it functions and how it's designed.

2.1. Event-loop structure



The image included shows the general design of the event loop. The event chain in any asyncio program must first begin by binding an event-loop to the currently executing thread. In modern versions of python, this amounts to calling the function asyncio.run(main_function). Following this call, an event-loop is bound.

Requests, as shown in the image, can be of various forms (in our prototype, requests were text-based requests from client executables, sent over TCP to our servers). These requests are then added to the event-loop. When an intensive request is being processed (such as I/O), the event-loop is free to move another task from the request pool to work on until the I/O task is completed. Once the I/O completes, the event-loop can then service the request that originally demanded that I/O. The key notion here is that the event-loop allows execution such that intensive requests outside the scope of the process do not block the process from continuing to execute.

3. Server Herd

In this section, we will begin a detailed discussion of the prototype project created to exercise the asyncio module's capability for servicing network I/O. The version of python used to create the server script is 3.9.2.

3.1. Goals

The overarching intent of this project was as follows: given a client and a herd of servers, implement a system such that the client can communicate geographic information via TCP to one member of the herd and get a request serviced by another member of the herd, irrespective of which server initially received the required information. This lends itself well to an asynchronous event-loop for a couple of reasons. Firstly, the propagation of information from one server to the rest is suited for an asynchronous framework, as the chain of events in a propagation may vary based on naturalistic factors. Secondly, client requests can be dynamic, sent to one server at one point and then quickly to another server at a different point: it makes sense that dynamic request sequences be mirrored in a server proxy that is equally dynamic.

3.2. Server connections

There are five servers associated with this project: Riley, Jaquez, Juzang, Campbell, and Bernard. When you call *server.py Riley,* the script will start the Riley server. The same is true for the other servers. The servers communicate with one another as follows: Riley communicates with Jaquez and Juzang, Campbell communicates with Juzang and Bernard, and Bernard talks with every server except for Riley. This communication is bidirectional. When one of the servers receives a request (the form of which will be detailed later), all the information needed to process that request must be propagated to the other servers. To handle this propagation, I initialized a coroutine called *propagate* which my code calls.

3.3. Requests

Requests are sent with a few viable formats. The request that starts the chain of events is of the following form:

IAMAT client_name client_coords time

The server that receives this request must then respond by saying:

AT server_name time_diff client_name client_coords time

In order to accomplish this, I used a dictionary that stores an association of a client_name to its coordinates. Thus, when a server receives a request from a client, it can get its coordinates easily. Additionally, I used dictionaries to store the last time that a client communicated and the last server that the client communicated to (as this information is also pertinent in the AT response). Clients can also send another request of the following form:

WHATSAT client_name radius items

This request from a client asks a server to utilize the Google Places API in order to find what places of interest are in a specified radius around a client's coordinates. This is an http request, which I made using the aiohttp module. The server responds with an *AT* request like before (it must contain information about the last time this client communicated and to which server) and the JSON formatted output of the Google Places API.

In order to propagate information about clients, the server that receives the *IAMAT* will send a request of another form:

PROPAGATE client_name client_coordinates time_information server_name

This ensures that every server that receives the propagation can then update its dictionaries to reflect the new information about the client, as well as continue to propagate along the chain. With no further steps, our implementation is incomplete: the propagation will go on forever. In order to cease a propagation, I checked the client_coordinates and time_information against what is already stored in the server's dictionaries: if they match up, we have already received the information from a previous propagation and we can stop.

Anytime a server receives a request that is formatted incorrectly, it responds with:

? request

Finally, servers also log their inputs, outputs, and connections to other servers. Some example logs are provided for the processing of an *IAMAT* to Riley, a *IAMAT* to Bernard, a *WHATSAT* to Bernard (of info sent to Riley), and a final *WHATSAT* to Riley (of info sent to Bernard).

3.4. Leveraging asyncio

As mentioned earlier, the dynamic nature of this task makes it prime for leveraging asyncio coroutines and await premises. The event-loop is tied to a main asyncio coroutine which awaits the "server.serve_forever" method: into this coroutine, I passed another coroutine which handles new connections to the current server. In this manner, each server can maneuver around multiple connection requests in an asynchronous manner. Within my connection handler coroutine, I call a few other coroutines depending on what the input to the server is.

In order to propagate asynchronously, I defined a coroutine that opens TCP connections with neighboring servers and sends the required information, which I *await* on: this is the keyword that lets the event-loop know that it can schedule other tasks while this one is completing. Propagation is an excellent candidate for asynchronous operation because, while a server is propagating, the event-handler can continue to run other pertinent tasks that are waiting in the event queue. On the flip side, while another server is engaged in a task, when it receives a request to continue propagation, it can do so dynamically, as the event-loop can interrupt the current task (given it is in position to be interrupted).

I also defined a coroutine for logging to a file (each server has its own log). While text is being written to the file, the event-loop should be free to schedule other tasks onto the operating thread. In order to accomplish this, I *await* on the coroutine responsible for logging. Thus, logging is another area in which asynchronous operation can be capitalized on to speed up execution.

Using coroutines for logging and propagation in combination with using appropriate await semantics contributes positively to the performance of this task. In comparison to multithreading where we might have to manage race conditions and lock semantics, asynchronous programming in a single-threaded environment can perform better due to reduced contention between threads and, additionally, is much simpler to code. Due to the prevalence of logging and propagation in this project, many servers are susceptible to interruption and incongruent orders of events, which is an area where asynchronous programming often shines. A more detailed discussion of how python language conventions affect programming asynchronously within it will be detailed later in the report.

3.5. Important caveats

There is one important, somewhat obvious, caveat that comes with the performance gain of asynchronous programming, and that is the lack of guarantee on any specific sequence of events within the event-loop. This should make sense to a reader following the discussion of asyncio that preceded this section of the report: by allowing tasks to, rather freely, interrupt other tasks and run to partial completion, we cannot enforce sequential operation on our code. If we wanted to do such a thing, our program would not be *asynchronous*. Regarding this project, one concerning order of events could be a server receiving a *WHATSAT* request for an *IAMAT* request that is still propagating to said server. This is theoretically possible in the asynchronous event loop as we await propagation; we cannot guarantee that the propagation finish by the time a *WHATSAT* is received because the event-loop is free to schedule coroutines such that they finish in orders we might not expect. With only five servers, this case is less likely, but still very possible. A way to combat this (at the expense of client response time) is to *await* for a small period of time, maybe a quarter-second or so, before a server that has received an *IAMAT* responds to its client. While the server is awaiting, the propagation has some additional time to complete. Therefore, we can lower the chances of this situation happening by a significant amount if we allow slightly worse response times.

3.6. Aside on asyncio versions

This python project was built on top of version 3.9.2. Therefore, asyncio functions like asyncio.run are used to bind the event-loop. These are functions not available in older versions of asyncio but, nevertheless, porting the project over should not be a very difficult task. New functions in modern versions of asyncio are more highly abstracted versions of lower-level functions like loop.run_until_complete or loop.run_forever (the source code for asyncio.run directly references these functions). Thus, the new asyncio functionality leveraged for this project can be replaced by the appropriate, lower-level asyncio functions that are present in dated versions of asyncio.

4. Viability & Java Comparison

In this section, details of the project, and asynchronous programming in general, will be examined in the context of python. More specifically, topics like dynamic typing, the global interpreter lock, and memory management within python will be discussed. I will also share how these language-level decisions affected my development of the server herd during my analysis of such areas.

Asyncio will also be compared to Node.js in JavaScript briefly, following the discussion of the above topics.

4.1. Dynamic typing

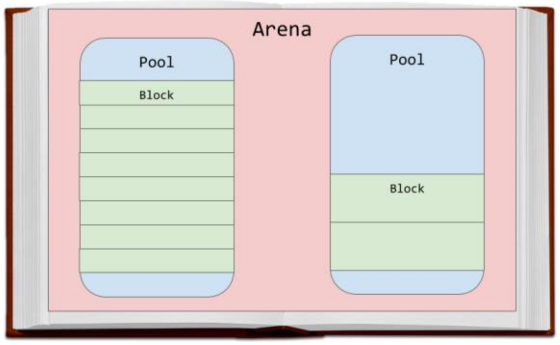
Python is a *dynamically-typed* language, meaning that there is no type-checking for declared variables. The type of a variable is determined by context and the assignment of that variable during runtime. This feature of python did not have any adverse effects on my completion of the server herd. In fact, the loose typing of variables made it easier to do certain operations like checking server input for bad formatting or logging input to text files. Ultimately, dynamic type checking made for an easier experience writing code as more detail could be abstracted out during the design process. The clear counterpoint to this is that runtime errors are more likely to happen in comparison to a *statically-typed* language, where variables must be declared with their types. Regardless, within my development of the asynchronous server herd, I cannot say that dynamic type checking was a negative: on the contrary, I would argue the opposite as it made my coding experience easier.

4.2. Global interpreter lock (GIL)

Processes can run with multiple threads under them, sharing a memory space. This is known as a multithreaded process. Python is inherently a single-threaded language due to a feature known as the *global interpreter lock*. The GIL is used to solve an issue related to garbage collection (or the lack thereof) in python. When a variable is initialized to reference an object in python, the *reference count* of that object increases by one. When that variable goes out of scope, the reference count for that object is decremented by one. Any object with a reference count of zero ceases to exist. The problem associated with reference counting and multithreading is one of simultaneous execution: one thread can decrement the reference count of an object to zero while it may still exist in the context of the other thread, leading to crashing and bad memory accesses. The GIL ensures that only one thread can run python's bytecode, which eliminates the potential to multithread but preserves the viability of reference counting. The biggest impact of the GIL with regards to asynchronous execution is that multithreading is impossible to do in python. Thus, the implementation of asyncio is tied to a single thread, which can lower peak performance.

While this might sound bad for asyncio, in programs with a lot of internal communication, such as a server herd that is constantly receiving and sending requests, asynchronous programming on a single thread can prove more viable than multithreading. To begin with, there is no need to handle race conditions or lock critical sections like you would with multithreading. More importantly, depending on the prevalence of I/O, this increased contention can prove seriously adverse for multithreading. In this context, not only can asyncio be easier to program with, but it can also offer better performance than multithreading. However, keep in mind that this is a gray area: there is no clear winner and for general tasks, multithreading generally fares better. Regardless, for tasks that involve heavy uses of I/O and internal communication, asyncio in python can perform just as well as multithreading would.

4.3. Python memory management



Within the memory management module of python, *blocks* of data are stored in *pools* of memory. Memory pools use linked lists in order to keep track of free blocks and unallocated blocks, which is more commonly known as a *free list*. On the highest level of abstraction, pools are organized in *arenas*, where pools with more free blocks take higher precedence over pools with less free blocks. Thus, the arena refers requests to pools, which will then refer a request to a specific free block, which will finally explicitly allocate the data.

As briefly mentioned before, python has no garbage collector within it and, instead, destroys allocated objects when the reference count of that object falls to zero. Thus, while managing memory is simpler in many ways, it is also more restrictive as it confines python to a single-threaded language. This choice of memory management made my coding experience simpler as I never had to worry about memory addresses or the allocation and freeing of objects as reference counting is a simple, autonomous process that happens smoothly in the background. However, in order to ensure that reference counting is viable, python is locked by the GIL and is singly threaded on the language level. Thus, modules like asyncio are always going to be single-threaded implementation, which can impact performance (but as I mentioned earlier, for tasks with heavy I/O and interruptive communication, maybe not so adversely).

4.3. Comparison to Node.js

In general, Node in JavaScript has more similarities than differences with asyncio in python. For example, both languages are dynamically typed, interpreted rather than compiled, and both implementations of asynchronous programming use similar functional premises, like *awaiting* and *generators* (analogous to coroutines in asyncio). However, there are some key language level differences between JavaScript and python that can affect performance and runtime.

The first notable disparity is the difference in garbage collection algorithms. Python utilizes reference counting while JavaScript has a dedicated garbage collector, which uses a *mark-and-sweep* algorithm in order to find garbage. This algorithm explores all references starting from the root referencing, marking all reachable objects currently allocated in memory. Then, all unmarked objects are removed. From the programmer's perspective, this should amount to little notable difference since these algorithms run on the lowest level. Thus, they rarely limit or meaningfully affect how programs are written in either language, though this might lead to some minor performance differences or other quirks beneath the surface.

The second, more notable, disparity is that JavaScript supports multithreading while python does not. Due to reference counting, python must be interpreted by a single thread whereas JavaScript is not limited in this way. This can lead to more disparaging performance differences: in some programs, multithreading can greatly benefit performance, especially in CPU-bound tasks where having multiple *workers* can be beneficial. However, this difference is not relevant to a comparison between Node.js and asyncio because both packages are **singly-threaded.** Thus, even though JavaScript might support multithreading and multiprocessing, when looking only at its implementation of asynchronous programming using node, it is comparable to asyncio's singly-threaded environment.

Ultimately, JavaScript differs from python in its ability to leverage multithreading and in how it deals with garbage memory using a dedicated garbage collector. While this may result in big performance differences outside of asynchronous programming, when comparing node and asyncio, these two language-level differences are largely irrelevant and will not lead to any language choice being better than the other.

5. Conclusion

As we reach the crux of the report, we come to the final question, perhaps the most important one, that must be answered: when building an application server herd which will receive constant updates of a dynamic nature, requiring copious amounts of internetwork and external communications during runtime, is asyncio a suitable solution? In light of the many things discussed in this report, such as how asyncio works on the software end, what features of asyncio can be leveraged to build a simple parallelizable proxy that interfaces with the Google Places API, and how asyncio compares to other asynchronous approaches amongst other languages, I would give an overall **pro recommendation** for asyncio. During the process of coding my project, asyncio made many operations, such as logging and propagation, modular and organized due to coroutines and await logic built into asyncio. Features like dynamic typing served not as negatives but positives in my coding experience and will have only minor performance impacts for a larger scale project. There are some concerns to be had, such as the lack of a deterministic chronology of events in an asynchronous framework, but this can often be solved if one is willing to accept sacrificing some response time on the client side. Thus, I would certainly recommend asyncio in python as a viable solution for building a larger, parallelized proxy for handling large amounts of Wikimedia queries: given the dynamic, event-driven nature of the task and asyncio's suitability for such things, I have no doubt that asyncio would be a strong solution for a project of such nature.

References

asyncio source code

https://github.com/python/cpython/blob/3.9/Lib/asyncio/runners.py

asyncio documentation

https://docs.python.org/3/library/asyncio-task.html#asyncio.run

Python memory management

https://realpython.com/python-memory-management/#memory-is-an-empty-book

asyncio python briefer

https://realpython.com/async-io-python/#the-asyncawait-syntax-and-native-coroutines

Python GIL information

https://realpython.com/python-gil/

Event-loop information + picture

https://eng.paxos.com/python-3s-killer-feature-asyncio