Asyncio for Application Server Herds

Faith Twardzik
University of California, Los Angeles
CS 131

Introduction:

Project 5: In Proxy Herd with Asyncio, we dealt with a potential service in which many new servers would need to be added and maintained (in the form of cell phones broadcasting GPS locations). Relying on the Wikimedia application server posed a problem, as it would be far too slow. Instead, the objective was to implement the idea of an application server herd, in which the servers can directly communicate with one another and a central server, allowing immediate transmission of data from a client to a server and from that server to the rest of the servers to which it is directly connected (in our application, this entails transmitting the GPS locations of the client cellphones in the form of IAMAT commands up to the server to which it has established a TCP connection). In essence, this should ensure an easy way of transmitting between servers, as well as a quick method for flooding all servers with new updates. In order to implement this server herd with IAMAT/WHATSAT functionalities, Python's asyncio networking library was utilized, the pros, cons, and details of which are the focus of the rest of this essay.

A Brief Introduction to Python Concurrency:

In order to contrast traditional methods of establishing parallelism or concurrency in Python with the concurrency of asyncio, we first define a few attributes of the Python multithreading/multiprocessing system. To begin, parallelism and concurrency are not equivalent. Both entail running parts of a program simultaneously in order to achieve speed-up. However, the former ensures that tasks are running simultaneously, by splitting up a task/s into subtasks, and then processing these on separate CPU cores for a single computer, or several computers in a network. This is the basic concept of multiprocessing. The latter – concurrency – does not guarantee that tasks are running simultaneously; it just provides a framework through which tasks *could* run simultaneously. Task execution is allowed to overlap, although it brings with it the need for locks or other such mechanisms to ensure that tasks are not causing data race conditions by accessing and updating memory out of sequence with other tasks, and producing incorrect results because of it. This is the basic idea of multithreading, where the tasks are split among threads.

In a Python-specific context, we must delve a bit deeper to understand how multithreading and multiprocessing can be implemented. The interpreter and the operating system both have a hand in creating and scheduling processes and/or threads (green and native threads, respectively). This suggests that Python allows multithreading, which it does. However, Python multithreading implementation is raucously inefficient, because of Python's Global Interpreted Lock (GIL).

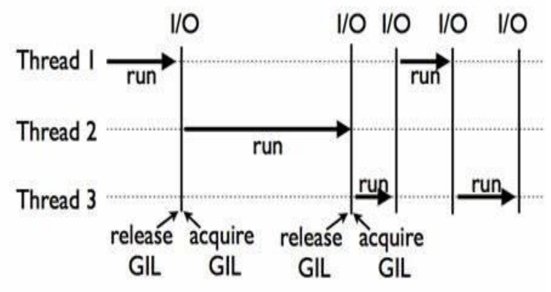

Figure 1 *Illustration of Multithreading with Python's Global Interpreted Lock*

The GIL runs at the interpreter level, locking Python objects so that only one thread can access them at a time. This is, in essence, sequential, where only one thread is allowed to execute in the Python interpreter at a time. Therefore, despite Python allowing multi-threading, the GIL makes it so that multithreaded programs act as if they are single-threaded, and are therefore extremely inefficient, sometimes less efficient than a program without multithreading, due to thread-maintenance overhead. However, we also note that while the Python GIL reduces multithreading to single-threading, ensuring only one CPU is used by a task at a time, this does not affect multiprocessing. In Python, programs are allowed to use multiple processes, with each running on a separate CPU processor, and so multiprocessing is much more efficient, and will allow parallelism in a way that multithreading in Python did not allow concurrency.

Asyncio In the Context of Concurrency

Multiprocessing connects to parallelism and multithreading to concurrency (despite parallelism being within the umbrella of concurrency, as parallelism implies concurrency), yet asyncio is defined separate of either. Asynchronous IO, the official terminology, causes concurrency, not parallelism. However, it is in no ways related to multiprocessing or multithreading, nor does it use any of the functionalities of these two methods. It is both single-threaded and single-processing. The mechanism for concurrency depends on what asyncio refers to as cooperative multitasking. The idea of cooperative multitasking, is that we can schedule tasks ourselves, and control when the CPU performs a context switch at the level of the program (to switch threads, as we can only run one thread on one CPU core), as opposed to having the Python scheduler do this for us, which is the case in multithreading operations. In multithreading, this context switching is what leads to data race conditions and inaccuracies of the program without mutexes/locks/semaphores to ensure multiple threads are not accessing/updating memory out of order or concurrently. Instead of using these devices, then, asyncio works by allowing us to manually choose when to context-switch, in a manner that will not allow mutual accesses to shared memory in the first place – thereby data races will never occur. Asyncio implements this using the idea of an event loop, which schedules async tasks, and the usage of asynchronous routines, also known as co-routines, which are called as such because they can pause (or yield, in earlier Python terminology) in the middle of execution in order to allow other co-routines to take their place and run instead, and then resume execution when the context is switched back over to them (from the scheduling that was specified in the event loop) to terminate execution. It is especially critical here to note that because asyncio manually context switches and can do so in the middle of asynchronous routines, this means that while there are sizeable wait times during the execution of a routine, a context switch can occur and allow other co-routines to execute in the meantime. This is the basis for why asyncio is optimal for I/O networking, as opposed to CPU-performance, because I/O operations include many tasks in which delays or waiting must be done. This is why asyncio should be used for I/O applications (such as our application server herd implementation), but should *not* be used in other applications that rely more heavily on the limitations of the CPU core being utilized – these limitations would be more efficiently overcome using multiprocessing. In essence, this proves that asyncio can easily run server herds, with exceptional performance, though for other applications seeking to implement simultaneous execution, asyncio is not necessarily the most optimal choice.

The Attributes Inherent to Asyncio

Ayncio utilizes an event loop to schedule its asynchronous tasks/co-routines. An asynchronous task, as mentioned, is able to halt mid-execution to allow other tasks to execute in the meanwhile. These tasks are built upon Python generators, which are functions that act somewhat like iterators, in the sense that they use *yield* in order to transfer control to the function caller, and then the caller, at any point within the execution of the function, can use *next* to continue execution (move the iterator forward). Yielding is what halts the program execution temporarily (while keeping a record of the stack at that point in execution), and it is this parallel that we note when we speak of

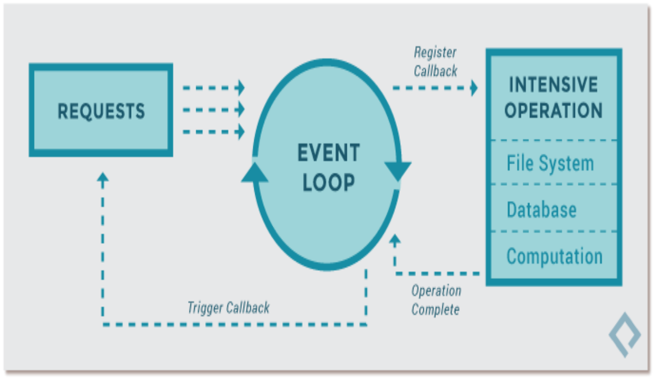

Figure 2 *Model of Asyncio's Event Loop Procedure*

asynchronous tasks and co-routines, which are basically using the concept of yielding in order to pause execution and context-switch. Notably, co-routines do differ from generators in that they can not only use *yield* pause functions and return values, but also pause functions and accept values. It is this added to the generator functionality that allows co-routines within the event loop to work asynchronously in the asyncio implementation. Next, we discuss awaitables, which come in the form of coroutines, tasks, and futures. These calls are marked manually using "await", so that when execution progresses to calling an awaitable marked by "await," such as "await asyncio.sleep()", the function pauses, abandons control back to the event loop, which has other tasks that could be scheduled, and allows the event loop to context switch to a different task while the awaitable is being awaited on. When the awaitable finishes, the event loop context switches once more, returning control to the current function, which resumes execution after the line with the "await" statement. Anything that includes waiting time, therefore, can utilize "await" to allow simultaneous execution to be carried out by other tasks/coroutines/etc. So, we have coroutines, tasks, and futures, which are all awaitable. Coroutines may be functions defined using the syntax "async def" or objects returned by these functions. Tasks schedule coroutines. One example of a task is the asyncio.run() function, which generates a new event loop and closes it after returning. It takes in as a parameter a coroutine, and schedules this coroutine to be executed, usually at the beginning of the program run. This coroutine is then transferred to loop.run_until_complete(), which takes in as a parameter a future. If we consider my program, I utilized this task in the following statements:

# within main, allowing the event loop to run forever, as long as it is not exited by manual input such as cntrl-c

asyncio.run(server.run_forever()) # here, server.run_forever() is an async-defined coroutine

The New vs. the Old

Newer versions of Python utilize native coroutines, while older versions of Python used a different syntax, which implemented generator-based coroutines instead. Generator-based coroutines are/were regularly defined function, with the added keyword @asyncio.coroutine. Instead of utilizing "await" in these coroutines, the syntax was "yield from". These generator-based coroutines have since become obsolete with newer versions of Python, because they were designed to be implicit, and therefore more easily confusable. As was noted, they shared syntax with ordinary Python generator functions, and in so doing lacked the originality that makes native coroutines more effectively distinguishable, as the keywords such as "async" for coroutine definitions and "await" for awaiting are exclusive to asyncio.

Another difference between older versions of Python and newer ones, is that program-level interaction with the event loop is abstracted-away by new built-in asyncio functions, such as asyncio.run(), as described in the previous section. Before Python 3.7, you would have been required to get the event loop, use asyncio.create_task() to create a task for it, and then transfer the newly-made task to the event loop. Now, this is superseded by the asyncio.run() function, which includes all three of these calls in one. It is far less error prone on one hand, introducing less programmer-error and opportunities to interact with the event loop in a way that might corrupt it. However, it also gives the program less low-level control over the event loop's execution. This is similar to the case generated by asyncio.create_task(), which in newer versions of Python abstracts away the need to manually accrue access to the task, create a future, and pass it to the task.

Asyncio vs. Node.js

Node.js is incredibly similar to asyncio, in that it also implements asynchronous programming, similar conceptually to asynchronous I/O for Python, though utilized for a slightly different application, and in JavaScript instead. Node.js is not a networking library, like asyncio, but rather a server environment that utilizes JavaScript. An example task in which Node.js would be utilized is in accessing a file on a server and transferring the contents back to a client (this has striking parallels to the HTTP request utilized in our application server herd implementation, which was retrieved and then transferred back to a client). Node.js will send the request for the file's contents up to the server. The server will then execute, and it is at this point that Node.js will be waiting to receive the contents. Just as asyncio utilized "await" to return control to the event loop and allow other coroutines to run while it was awaiting a task, Node.js does the same. While it is waiting, it switches context, so that new requests can be passed in and processed while it is waiting to complete the first request. After the server reads the file, retrieving its contents, control returns back to this procedure so that the contents can be referred back to the client. This is in much the same way as control switches back to the current Python coroutine after the call that was awaited returns, so that the coroutine can continue to execute.

Miscellaneous Topics: Memory Management, Type Checking, Implementation & More

Python utilizes dynamic typing, so that type checking occurs while the program is running. This is in opposition to statically typed languages, with static type-checking, such as C and Java, in which the compiler performs type-checking before the program is run. In Python, this means that a variable's type may also be modified over the course of a program run. Python dynamic typing allows the programmer more flexibility in type usage, therefore, but this transfers to a greater risk for programmer error in types, which will only be discovered while the program is running (and will likely produce runtime logic errors), as opposed to the compile-time errors that you would receive with C and Java, that would disallow you from running a buggy program. Python also utilizes what is known as "duck typing," where types are not checked at all, but rather the existence of a definition for a method that is called on an object is checked for. In terms of memory

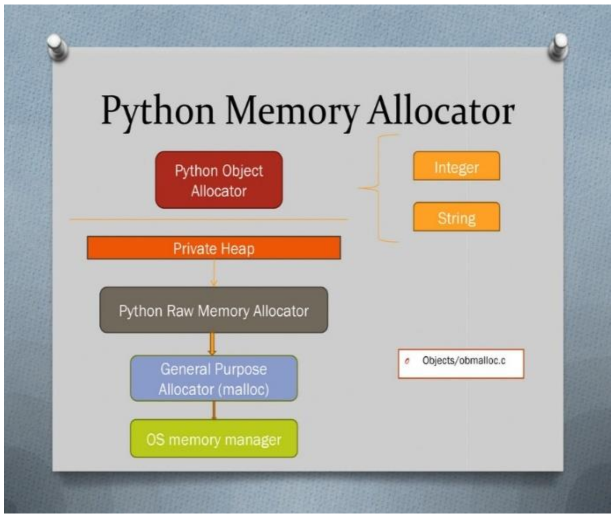

Figure 3 *Python Memory Allocation Using the Memory Manager*

management, Python utilizes a garbage collector. Whereas in languages such as C and C++, dynamic memory allocation and freeing was the task of the programmer utilizing functions such as malloc(), realloc(), and free(), in Python, the interpreter performs these tasks instead. In fact, Python implements a memory manager, which maintains a private heap. This heap stores all of the objects, data structures, etc. that are in use in a program, and the memory manager is tasked with dynamic memory allocation, freeing, segmentation, etc. Since it is the interpreter (through the memory manager) that has access to this heap, the programmer does not. The Python memory manager acts as a middleman, where a task such as allocating dynamic memory would be passed to the memory manager to do. In addition, Python utilizes a garbage collection system with a reference count algorithm, in which a reference count is maintained for every object, and when this reference count drops to zero, its memory is freed. The second algorithm used is for the case in which circular references occur. Because of this middleman system, Python memory management is abstracted from the programmer, less error-prone, easier from a programming perspective, and unable to cause common error such as memory leaks. However, it is also highly inefficient (the garbage collector takes seconds at a time to round up and free objects with reference counts of zero), and allows the programmer less control over internal functions.

We conclude with a few implementation problems that I ran into. One of these was dealing with the case in which a server connection was dropped. My flooding algorithm utilizes recursion to visit all of the servers connected to the server we are currently processing updates for, so when it attempted to call open_connection() for the server that had gone down, the entire program run hit an exception and crashed. I fixed this by utilizing a try/except statement. Another issue I hit was in processing server-to-server communication bidirectionally, and transferring not only the flooding message, but also the client's attributes, over to each server. This, I am still unsure of if I implemented correctly, as it was difficult to test. Besides this, I had troubles limiting the results of the Nearby Places search response, as well as wrapping up the program in checks for invalid inputs and crashes that might occur if the input was unexpected, and I am sure there are many cases of invalid input for which my program may have undefined behavior, although the vagueness of the spec in issuing directives for how to deal with these cases leads me to think this will be common to most programs.

References

"Concurrency vs Parallelism." *Tutorialspoint*, www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_concurrency_vs_parallelism.htm.

"Coroutines and Tasks¶." *Coroutines and Tasks - Python 3.8.3 Documentation*, docs.python.org/3/library/asyncio-task.html.

McDonnell, Mark. "Guide to Concurrency in Python with Asyncio ⋆ Mark McDonnell." *Integralist*, 30 Nov. 2019, www.integralist.co.uk/posts/python-asyncio/.

"Memory Management¶." *Memory Management - Python 3.8.3 Documentation*, May 2020, docs.python.org/3/c-api/memory.html.

Noronha, Bosco. "Multithreading vs Multiprocessing in Python 🐍." *Medium*, Noteworthy - The Journal Blog, 6 May 2019, blog.usejournal.com/multithreading-vs-multiprocessing-in-python-c7dc88b50b5b.

Pundir, Pawan. "Threading vs Multiprocessing in Python." *Medium*, Practo Engineering, 9 Aug. 2017, medium.com/practo-engineering/threading-vs-multiprocessing-in-python-7b57f224eadb.

Solomon, Brad. "Async IO in Python: A Complete Walkthrough." *Real Python*, Real Python, 29 Apr. 2019, realpython.com/async-io-python/.