Application Server Herd: Python and Asyncio

Anthony Khoshrozeh

ABSTRACT

In this paper we look at an alternate architecture for a web server called an Application Server Herd. The server is implemented with Python and the `asyncio` library. This paper will look at the suitability of implementing such an application with Python and `asyncio`, its performance implications, comparing the intrinsic features of Python and Java, as well as `asyncio` and Node.js.

1. INTRODUCTION

Wikipedia and its related sites are based on the Wikimedia server platform, which is based on GNU/Linux, Apache, MariaDB, and PHP+JavaScript, using multiple, redundant web servers behind a load-balancing virtual router and caching proxy servers for reliability and performance. We want to build a new Wikimedia-style service designed for news, where (1) updates to articles will happen far more often, (2) access will be required via various protocols, not just HTTP, and (3) clients will tend to be more mobile. In this new service the PHP+JavaScript application server looks like it will be a bottleneck. From a software point of view our application will turn into too much of a pain to add newer servers (e.g., for access via cell phones, where the cell phones are frequently broadcasting their GPS locations). From a systems point of view the response time looks like it will too slow because the Wikimedia application server is a central bottleneck.

To address these problems, we are investigating the `asyncio` library in Python and determine if such would be an effective and practical alternative architecture. To do so, we are implementing a simple and parallelizable proxy for the Google Places API.

2. ASYNCIO

2.1 Overview of asyncio

`Asyncio` is a Python library used to write asynchronous, concurrent code using the async/await syntax, which is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc [1]. Using an asynchronous code allows us to let other computations to occur while other code is waiting on something like I/O from the network. To understand the suitability (pros and cons) of using such library, we'll go a few of the key features of the library.

2.2 Coroutines and Tasks

Coroutines are how we write asyncio applications. A coroutine can be thought of as objects similar to threads but cannot run in parallel. However, they are run asynchronously. When a coroutine is called, it is not scheduled immediately. To execute it we have 3 different mechanisms. First is calling the asyncio.run() function, which runs the passed coroutine, taking care of managing the asyncio event loop, finalizing asynchronous generators, and closing the threadpool [2]. Second is the using the `await` keyword, which makes the coroutine awaitable and can be awaited from other coroutines. The await keyword suspends the execution of the current coroutine until the awaited function is finished. Coroutines can also be executed by calling the `asyncio.create_task()` function to run coroutines concurrently as `asyncio` Tasks. Wrapping the coroutine in the Task allows it to be scheduled to execute when the Task is executed inside an Event Loop.

2.3 Event Loop

Event Loops are at the core of every asyncio application. Event Loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses. They are lower-level code that provide finer control over event loop behavior (as opposed to using `asyncio.run()`). The Event Loop uses cooperative scheduling and allows us to run tasks that are waiting to be executed. They can be used instead of using `asyncio.run()`. Event Loops also allow us to create low-level APIs for network I/O, such as calling the `asyncio.start_server()` function, which starts a server that accepts TCP connections.

3. SUITABILITY OF ASYNCIO

3.1 Pros: Asynchronous Code

One of the most important aspects of using asyncio and a large part of why it makes this library a good choice for implementing an application server herd is that it allows us to run our code asynchronously. The application we want to design needs to be able to handle frequent updates from clients and other servers in the herd. Asyncio's key

features, coroutines and event loops, allow us to asynchronously handle these updates without creating a bottleneck like in Wikipedia's architecture. Using synchronous code would create a long queue of tasks to be run, since the code is executed sequentially, as server resources become in higher demand as the clients increase. By allowing other coroutines to run while others that are waiting for network I/O block, we can minimize task starvation and greatly increase the efficiency and workload of our application.

Also by having servers flooding (a coroutine), we can update our servers without having to go through a central application server, which creates a bottleneck under heavy loads.

3.2: Pros: Ease of Writing Applications

Python has an intuitive syntax and is allows programmers without much or no experience writing Python code to quickly develop applications. It also has powerful semantics, allowing developers to write code that might take 10-15 lines in a language like C or C++ in just a line or two.

The asyncio library is also well documented and provides lots of examples on how to use its APIs and libraries, which greatly speeds up development time. Outside of official documentation, there are also many examples and tutorials how to use the asyncio library and almost every other Python library since it is one of the most popular languages being used today to develop software. Having a large developer community around a language is very important for maintain code bases and fixing bugs.

For these reasons I found writing the proxy herd not to be very difficult and was able to do so rather quickly compared to other projects.

3.3 Cons: Inner Workings of Python

While the asyncio library lets us run code concurrently, we cannot write code that runs parallel, which is a key difference. This fact doesn't stem from the library but the inherent design of Python, which the effects of this will be discussed in the comparison of Python and Java (see section 4).

4. PYTHON VS. JAVA

4.1 Type Checking

Python uses dynamic checking which allows to use develop applications quicker since we don't to be keep track of our data types as carefully as in a language such as C++ or OCaml. However, this often can result in runtime errors that would have been prevented if we had used a language that has static type checking. This means in order for us to increase the reliability of our code, it needs to be thoroughly tested, and most likely more so than a statically type checked program. The benefit of this is a simpler syntax and more flexible code.

Java, however, uses static type checking, so it will catch errors at compile time rather in runtime like Python. This can prevent many, possibly disastrous, errors that only happen in very rare case for which would have remained undetected for a while if our program used dynamic type checking. Static type checking also means our code has to be explicit in our we define our objects and structures and leads to a more complex syntax. For an application of this size (only a couple hundred lines of code), using Python is suitable in this aspect, since it's small enough to examine carefully and catch any errors made.

4.2 Memory Management

Python has automatic memory management that uses reference counting to keep track of the times an object is being used. When the reference count is zero, the memory is freed. It is a simple approach and allows programmers not having to declare when they need memory or are done with it. This comes at a cost too, since there is some overhead associated with updating the reference count for all our objects. Also, if there is a reference cycle in the program, memory management won't run garbage collection on those objects even though they aren't being used, which means the program has a memory leak. This means over time if there are a significant number of leaks, the server must be restarted.

Java uses a generation-based garbage collection alongside the mark and sweep algorithm. "The heap is sometimes divided into two generations called the nursery and the old space. The nursery is a part of the heap reserved for allocation of new objects. When the nursery becomes full, garbage is collected by running a special young collection, where all objects that have lived long enough in the nursery are promoted (moved) to the old space, thus freeing up the nursery for more object allocation. When the old space becomes full garbage is collected there, a process called an old collection" [3]. The sentiment behind this approach is that most objects are short lived so partitioning our allocated objects gives us a more efficient and effective method of garbage collection.

Python uses a much more simple approach to its memory management than Java. This leads to Python having more memory leaks and also ends up much slower due to its constant reference counting. So, Java takes the lead over Python for memory management.

4.3 Multithreading

Python does not allow any race conditions to occur due to its memory management system of reference counting. If there were race conditions, this would lead to more memory leaks or releasing memory that is being used. Python uses a single lock, the GIL, to prevent deadlocks which would arise from using many locks on different objects. "The GIL is a single lock on the interpreter itself which adds a rule that execution of any Python bytecode requires acquiring the interpreter lock. This prevents deadlocks (as there is only one lock) and doesn't introduce much performance overhead. But it effectively makes any CPU-bound Python program single-threaded" [4]. This means Python code runs single-threaded code fast (less overhead from having multiple locks) and multithreading doesn't improve CPU intensive tasks. However, Python is multithreaded. It just switches between the threads instead of running them both at the same time.

Java, however, does support multithreading in the sense that multiple threads can be ran at the same time (in parallel, unlike Python which runs concurrently; sharing the CPU). Java was designed to multithread safe, using the `synchronized` keyword to prevent race conditions from occurring. This is a huge benefit, as parallelizable code offers tremendous performance boost.

So, Java can run multithreaded code much faster than Python since it is actually parallelizable and Python can only run concurrently.

5. ASYNCIO VS. NODE.JS

Node.js is "an asynchronous event-driven JavaScript runtime, …, designed to build scalable network applications" [5]. It is quite similar to Python's asyncio in that Node.js also uses an Event Loop to run asynchronous code. "The event loop is in the heart of Node.js / Javascript - it is responsible for scheduling asynchronous operations" [6]. They also both run code concurrently and not in parallel. Asyncio uses coroutines and Node.js uses callbacks to serve essentially the same functionality.

Even though they are quite similar, Python and asyncio are a more reliable choice than Node.js, even though Node.js offers a better performance on asynchronous code (Node.js is based off Chrome's V8 engine). Also you easily build your front-end in Javascript which means an easier time transporting data between the back-end. If your application needs security more than speed, asyncio is the better option. If you're trying build something with where speed is more important, Node.js will probably be the better option.

6. CONCLUSION

To conclude, using Python and asyncio to implement the application herd server is quite a good option with several benefits to replace the Wikimedia architecture. Even though it won't parallelizable, it'll run concurrently due to asynchronous code which will greatly reduce the bottleneck experienced in the previous architecture. Java has some benefits as explained above, but Python and asyncio has real notable pros too, such as ease of development. Also,if we're creating a herd server with not 5, but several hundred servers, using asyncio over Node.js is more suitable since it offers greater reliability. Overall, it is a very suitable framework to build such an application.

7. REFERENCES

[1] Python3 Asyncio Documentation:

https://docs.python.org/3/library/asyncio.html

[2] Python3 Asyncio Task Documentation:
https://docs.python.org/3/library/asyncio-task.html#running-an-asyncio-program

[3] Oracle Docs on JMM
https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html

[4] What Is the Python GIL?

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

[5] https://nodejs.org/en/about/

[6] https://blog.risingstack.com/node-hero-async-programming-in-node-js/