Proxy Herd with `asyncio`

Amy Seo

Abstract

In this project, we will be exploring asyncio, Python's asynchronous networking library, as a potential option for replacing the current Wikimedia server platform. More specifically, we will be seeing how asyncio works in a server herd and the reliability and maintainability of this library through the creation of a simple, parallelizable proxy herd.

1. Introduction

The current Wikimedia server platform is based on Debian GNU/Linux, with many web servers in the Linux Virtual Server load-balancing software. There are two layers of caching proxy servers in a complex infrastructure that is all hooked together. In this project, we are testing the waters for a new service where updates to articles would happen more often, access will be required through many different types of protocols other than just HTTP/HTTPS, and clients are more likely to be mobile. The current Wikimedia server platform uses PHP+JavaScript for their application server, making it difficult to add new servers and having much slower response times, making the Wikimedia application server a central bottleneck.

We will be looking at a different infrastructure, the application server herd, which has multiple application servers communicating with one another. This will be more useful for our news service as it is designed to keep up with rapidly changing data through the use a of stable database and interserver communications. To conduct research on this architecture, we will be focusing on Python's asyncio asynchronous networking library. Asyncio is event-driven and seems suitable for our rapidly changing data at large volumes. This infrastructure works by allowing servers to talk to one another and flood data back and forth without the current bottleneck that exists in Wikimedia.

2. Asyncio Framework

2.1 Pros

Asyncio is extremely compatible with other Python packages, and the learning curve is extremely low. The integration with aiohttp to utilize the Google Place API was extremely smooth, and its methods were perfectly usable even when in an asynchronous method. This most likely applies to other Python packages since it is pretty easy to utilize them in general. Being able to use these libraries within the asynchronous coroutines is definitely a plus.

Additionally, Asyncio utilizes an event loop that is constantly checking for requests, and when a request is made, it performs the corresponding coroutine. Because of this, it allows for asynchronous event handling during the entirety of the server's lifetime. Coroutines are specifically used in asynchronous programs because they are able to give up control when needed to allow more than one application to run at the same time. Due to its asynchronous nature, our server is then able to shift between functions without having to wait for methods to be run in order. This is extremely helpful for our project because we would be able to switch between functionality that is not being used to other ones that are.

2.2 Cons

One of the biggest disadvantages of asyncio is its lack of multithreading, that is, it is a completely single threaded framework. Because of this, there is no parallelization that can occur. Instead, there is just a singular event loop that can stop, pause, and start various coroutines. The single-threaded character of asyncio could allow for bottlenecks in the program, which is already a problem being dealt with in the existing Wikimedia server. Therefore, asyncio could never be as fast as other multithreaded frameworks no matter how efficiently the coroutines are handled. Since our simple proxy herd used a handful of servers and simple coroutines, the extent of this efficiency problem could not be robustly tested. If asyncio was to be used for a much larger network, I would imagine a lack of scalability compared to other multithreaded frameworks.

2.3 Ease of Use

Asyncio was pretty simple to use. It was very similar to running any other Python program, and since Python is relatively easier to write a program in compared to other languages, there wasn't much difficulty. Also, there is a lot of documentation on asyncio so it was easy to learn about how to create an event loop and start a server, and any questions were easily answered with a simple Google search.

2.4 Performance

Since asyncio runs on Python, this language choice is already a performance disadvantage because Python is an interpreted language that would take more time to run compared to a compiled language such as C/C++.

Although asyncio is pretty simple to use, it needs many external packages to create a useable server. In our simple

proxy herd, we had to utilize aiohttp to make an HTTP request, which is something commonly done in servers and so I was surprised that was not a functionality included in the framework.

As mentioned previously, the performance of asyncio is much lower than other multithreaded frameworks because it is single threaded. If the project is going to be large with many servers and connections, it would be much better to go with a framework that supports multithreading.

2.5 Python 3.9

It is not very important to rely on asyncio features of Python 3.9 and later because the necessary functionalities are able to be run on older versions of Python. However, there are certain functionalities such as support for an attempt at multithreading with asyncio.to_thread() that is included in Python 3.9. However, it is not necessary as mentioned above.

3. Python vs. Java

3.1 Type Checking

Python is dynamically typed while Java is statically typed. Since Python is an interpreted language, type checking is done at runtime. Although this lets the programmer have an easier time writing code without variable labels, the interpretation that occurs at runtime takes up a lot of time, and unexpected type errors could occur during runtime, causing unforeseen problems. In contrast, Java utilizes static typing, which means that the programmer has to take more time writing variable labels and making sure that all works at compile time. However, this allows for a more stable program as any problems could be checked and fixed at compile time. Therefore, when programs need to work perfectly, it is important to choose a statically typed language to make our program completely reliable no matter the circumstance, instead of worrying about possible runtime errors with dynamically typed languages.

3.2 Memory Management

Both Python and Java have their own memory management system called garbage collection. There is an internal system that keeps track of memory usage and allocates and frees objects on its own without any explicit memory calls from the programmer. Although this removes the possibility of dangling pointers and memory leaks, there is a performance cost.

Python uses a “reference counting” method where the number of references to each object is kept track of. Whenever an object’s reference count is 0, the garbage collector will mark and free the object.

Java uses a mark and sweep method as well, but does not count references like Python. Instead, it checks for “reachable” code in the program. When there is code that can no longer be reached, it frees it. Compared to Python’s reference counting, this is faster but takes up more memory.

Since our project has rapidly changing data and would need to process a lot of requests, Python’s reference counting method would be preferable because it would use a lot less memory compared to Java’s method.

3.3 Multithreading

As mentioned above, Python and asyncio unfortunately does not support multithreading. Although there were some updates to asyncio in Python 3.9 as mentioned earlier that attempts at some workaround to this lack of multithreading problem, there is no real multithreading. In fact, the language as a whole does not support multithreading due to the “Global Interpreter Lock” (GIL) in CPython, which restricts Python to use only one thread. Basically, there is no support for true multithreading.

In contrast, Java has true multithreading and a lot of built in functionality such as threads and locks. Therefore, Java is much more advantageous when it comes to multithreading and making programs more efficient.

4. Asynchio vs Node.js

Node.js is a JavaScript runtime environment that allows for asynchronous event handling. JavaScript has a lot of event-driven and asynchronous features that allow for asynchronous programming, and is an extremely popular choice in modern times as JavaScript is the most used language for web applications.

As mentioned earlier, asynchio is a Python framework that allows asynchronous server programming. Although Python is a popular language, JavaScript is heavily used in the web application scene, and could be used for both frontend and backend applications. Since Python is not inherently event driven or asynchronous on its own, the usage of Node.js seems much more beneficial because of the asynchronous nature of JavaScript.

It would be worth looking more into Node.js, as it allows for both front-end and server-side code to be written inn one language, and there are a vast number of libraries with npm that could have more potential than asyncio.

5. Conclusion

Overall, I recommend Python’s asyncio for the development of the application server herd due to its effectiveness in storing and propagating information across a network of servers. Even though there are some cons compared to other possible choices such as Java or Node.js, asynchio is able to fulfill the needs of a Wikimedia-style service designed for news. Additionally, it’s ease of use would make it preferrable for the programmers as python is a much more simpler language to write with, and the technicalities of using asynchio does not differ much from the usual python

program. In conclusion, asynchio would be a reliable framework to use for this project.

References

(1) *Asyncio — Asynchronous I/O — Python 3.10.4 Documentation*. https://docs.python.org/3/library/asyncio.html. Accessed 28 May 2022.

(2) Yellavula, Naren. "A Minimalistic Guide for Understanding Asyncio in Python." *Dev Bits*, 10 Jan. 2022, https://medium.com/dev-bits/a-minimalistic-guide-for-understanding-asyncio-in-python-52c436c244ea.

(3) Python, Real. *Async IO in Python: A Complete Walkthrough – Real Python*. https://realpython.com/async-io-python/. Accessed 28 May 2022.

(4) "Memory Management in Python." *GeeksforGeeks*, 30 Apr. 2020, https://www.geeksforgeeks.org/memory-management-in-python/.

(5) "Java Memory Management." *GeeksforGeeks*, 13 Dec. 2018, https://www.geeksforgeeks.org/java-memory-management/.

(6) Python, Real. *What Is the Python Global Interpreter Lock (GIL)? – Real Python*. https://realpython.com/python-gil/. Accessed 28 May 2022.

(7) "Introduction to Node.Js." *Introduction to Node.Js*, https://nodejs.dev/learn. Accessed 28 May 2022.