Python Proxy herd with Asyncio

Aris Luk, *UCLA CS 131*
UID: 905326942

Abstract

Python has risen to be one of the most used programming languages in the world, being applied to many different applications and use cases. This paper explores the use of the asyncio package in Python as a way to set up and run a server herd network. In this project, we try to fix the Wikimedia server model, which has a bottleneck in the application server. We will use asyncio to set up a network similar to the one produced on the Wikimedia platform and go over its advantages and disadvantages.

1. Introduction

Wikipedia and other Wikimedia server based websites run their application servers through a Linux Virtual Server load-balancing router, which works well for encyclopedia sites with fewer HTTP requests, as more info is delivered on a single page and users will spend more time on each page. However, our wanted application server should be able to support quickly updating pages and the increasing number of requests from users, like GPS updates from mobile phones and the frequent access of quick-read news sites. The Wikimedia model has an architectural bottleneck in its virtual router, which can only support so many requests in a short time period. In turn, we set out the develop a new architecture, the "application server herd", using the Python package asyncio to better handle large volumes of requests and updates. This new system will work by enabling the servers in the network to communicate with each other and "flood" data to each other without the use of a load balancer. This removes the Wikimedia bottleneck, and in our specific implementation, we are focusing on propagating GPS info within a network of servers.

2. Is Asyncio a Suitable Framework?

2.1. Pros

Asyncio, as its name suggests, enables asynchronous events to occur throughout the runtime of the program. This is done through the use of a program event loop that we set to repeatedly check for requests and process them by running them through the correct coroutines. Coroutines are similar to subroutines, but are used in asynchronous programs since they make it possible for the process to "voluntarily yield (give away) control periodically or when idle in order to enable multiple applications to be run simultaneously" (1). This is suitable for our architecture since it makes it easy to switch between methods on the fly without having to run everything in order. This is beneficial to our goals since we will be able to pause coroutines that are not currently in use, like a request parser, do other operations to keep the server herd operating, and restart the coroutine if a new request comes in.

Another advantage of using asyncio is its compatibility with other Python packages and the language itself. In my implementation of the application server herd, I used aiohttp to get data from Google Places since asyncio does not have its own methods to do this. The integration worked seamlessly, and aiohttp methods were operational even when nested in an async coroutine. I would expect this to be the case for many of the other packages in the vast Python library, which would be beneficial to the ease of implementation of asyncio, since it should be easy to add functionality through packages. The seamlessness of being able to use async, await, and yield keywords to denote coroutines and their operation are also an advantage to using asyncio.

2.2. Cons

A major disadvantage of asyncio is its single-threaded nature. At the core of its operation, asyncio programs are really just one event loop that runs through different coroutines, just that coroutines can be paused and resumed to let other coroutines run in the time between. There is no parallelization or multithreading available to us through the package, which could be a major bottleneck to its performance. Even if the yielding and resuming of coroutines are optimized to be as fast as possible, at the end of the day there are still a certain number of operations that have to be done on a single thread, which has a ceiling of how fast it can finish. This can be a problem for our large scale implementation of the server herd since it won't be as scalable as a multithreaded language option.

2.3. Ease of use

Asyncio is relatively easy to use in our application of a server herd. The core format of the code follows normal Python code closely, just with added async and await keywords for the defining and usage of coroutines. It is also easy to open connections between servers, using the open_connection method, and control the main event loop using the asyncio functions. When writing the implementation of the server herd, it did not diverge much from the normal process of writing a Python program.

2.4. Performance

As mentioned in the cons section, asyncio can suffer from performance issues as a result of its single-threaded nature. When compared to other options that can support multi-

threaded operation, it may be preferable to go with a multi-threaded solution if we want to handle a large number of requests in a short amount of time.

Another aspect of performance that we may have to worry about is the amount of packages that is required to write an operational asyncio program. Since the asyncio package itself doesn't come with a lot of features outside of its core function, we may have to import a lot of Python packages in order to fulfill our goals. For example, in this project we had to import aiohttp and json for HTTP requests and processing. Additionally, the fact that asyncio runs on Python, and interpreted language, is already a performance disadvantage compared to lower level languages like C++.

2.4. Python 3.9 Features

It is not very important to rely on Python 3.9 asyncio features, as older versions of Python already include the needed functions to create the application server herd, such as open_connection and the normal event loop handling methods.

2.4. Implementation Problems

The main struggle I had with my implementation of the server herd was figuring out how to flood the network with a new IAMAT message without it going into an infinite loop. I first started with a simple open_connection call to connect servers and propagate data, but I found that the flooding would continue in a loop. I solved this problem by utilizing the time information in the input message, and only continued flooding if the message was not already in the client list or if it was a message with a more recent timestamp.

3. Python vs Java

3.1. Type Checking

Type checking is a major difference between Python and Java. In Python, variables are dynamically typed, which means that Python programs "perform type checking at runtime" (3). This means that when we are writing Python programs, we do not have to worry about labeling our variables to be the right type, as the interpreter will automatically figure out what works with our implementation. Python's type checking is known as Duck typing, as the interpreter will check for methods and attributes in variables over an explicit type name. However, Java does not have dynamic typing. In Java, all variables are statically typed, which means when writing Java code, all variables have to be labeled before the code can be run. This can be a tedious process for the programmer, as they have to go through all their variables and make sure everything works with each other. When Java code is compiled down to bytecode, the compiler will check each line in the program to make sure the types are compatible and won't cause any type errors during runtime. Although this is a very safe method of typing, it puts more work on the programmer and makes source code more complex and heavier.

For the pros and cons of each implementation of type checking, Python's is faster to write and understand, but its dynamic type checking can lead to unexpected type errors during runtime if there is an edge case. However, Java will not have these type checking problems during runtime because they will be resolved during compile time. For our implementation of a large scale Wikimedia-type application server herd, it is preferrable to have a more stable program, so Java would be advantageous in type checking.

3.2. Memory Management

The major difference between the memory management techniques of Python and Java are its garbage collection methods.

In Python, a method called "reference counting" (5) is used, which keeps a count of references to each object in the program. When the reference count falls to 0, the garbage collection system, a "non-moving mark and sweep" system will mark objects and free them. This makes Python's garbage collection system very quick in freeing objects that are no longer in use.

On the other hand, Java uses a similar mark and sweep method, but instead of using reference counters, it checks for "reachable" code in the program. It will relocate code that can no longer be reached and free the collected code in one go. This is faster than Python's approach, as it has been optimized a lot throughout the lifetime of Java, but takes up a lot more memory space.

With respect to our project and application of a server herd, our program wants to process a lot of requests quickly, which will require a lot of memory, so the Python approach would be advantageous since it requires less memory for garbage collection.

3.3. Multithreading

Python does not have support for multithreading because of CPython's "Global Interpreter Lock" or GIL, which "allows only one thread to run Python code at a given time" (1). So even if our servers can have multiple threads, the GIL prevents multi-thread concurrency.

On the other hand, Java has a deep support for multithreading. Java has classes built for multithreading use, like "thread", and also has locks and other objects that enable a program to be parallelized.

In terms of multithreading, it is clear that Java is advantageous since it has built in support for it, while Python does not have multi-threading support, which requires us to use packages like asyncio.

4. Node.js vs Asyncio

Firstly, Python's asyncio is a Python package that brings in methods and features that enable asynchronous server programming, while Node.js is a runtime environment for Javascript that works with asynchronous server programming.

Asyncio's advantages include being friendly to new developers and easier to use, as well as better documented. This is a result of Python being one of the most popular languages in the world. On the down side, asyncio has problems with multithreading and can suffer from some performance issues as mentioned earlier in this paper.

For Node.js, it uses JavaScript as its coding language, which enables seamless integration with web development, since most web applications are written in JS. This enables developers to write back-end server code and front-end interface code in the same language, reducing the number of technologies required to get something like our news site online. However, there are documentation problems with Node.js, and its quick growing pace reduces the quality and availability of documentation for all of the new updates to the framework.

Ultimately, I think sticking with asyncio is good because of its well documented state and ease of use. I was able to get the servers up and running in a reasonable amount of time and the syntax was friendly to new developers, so asyncio is sufficient for our use case.

5. Conclusion

In conclusion, I would recommend Python's asyncio to my boss for the development of the application server herd. Although asyncio has some disadvantages compared to other languages, like lacking the multithreading and safety of Java or the front and back-end integration of Node.js, it is serviceable for our project of creating a network that stores and propagates location information.

References

(1) Velotio Technologies, "An Introduction to Asynchronous Programming in Python", medium.com/velotio-perspectives/an-introduction-to-asynchronous-programming-in-python-af0189a88bbb

(2) Robinson, Scott, "Python async/await Tutorial", stackabuse.com/python-async-await-tutorial/

(3) Oracle, "Dynamic typing vs. static typing", docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html

(4) Zomaya, David, "Python vs Java: Which Programming Language is Right for You?", blog.udemy.com/python-vs-java/

(5) Memory Management Reference, "Memory management in various languages", www.memorymanagement.org/mmref/lang.html

(6) STX Next, "Python vs. Node.js: Comparing the Pros, Cons, and Use Cases", www.stxnext.com/blog/python-vs-nodejs-comparison/