Python's Asyncio Library: A Potential Framework for an Application Server Herd

Abstract

Python's asyncio library could be used to implement an application server herd since it offers the ability to write concurrent code with cooperative multi-tasking. This would allow communication between servers to be quick since the program would not idle when doing I/O, but concurrently run other tasks instead of wasting CPU time. However, there are limitations with Python itself. It uses dynamic type checking when static checking would be a better option for this type of application especially for bigger applications. It also uses reference count for garbage collecting which has high performance, but cyclic garbage will never be collected.

1 Introduction

An "application server herd" is an architecture where multiple application servers communicate directly with each other. These servers are used to communicate rapidly-evolving data between each other whereas databases are used for less-often accessed data. When one server receives data, it will forward this data to the servers it is connected to and these servers will forward the data to any servers they are connected to. So each server in the herd will receive the data quickly without having to communicate with the database.

Python's asynchronous networking library, asyncio, is a possible way to implement an application server herd. The library offers the ability to write concurrent code without multithreading or multiprocessing. Instead, it uses one thread and one process to do cooperative multi-tasking. Concurrency is the idea of having multiple tasks running in an overlapping manner, while cooperative multi-tasking is where tasks voluntarily take breaks and let other tasks run. The use of concurrency and cooperative multi-tasking should allow the servers in the server herd to not waste time idling while waiting for I/O and instead servers should be able to process an update and forward it to other servers quickly.

However, the limitations of using Python for this type of application must also be considered. Python uses dynamic type checking, a heap to store Python objects and data structures, reference counting for garbage collecting, and its implementation of threading does not offer all of the features that Java's threading model does. This could cause problems for larger applications and make a Java-based approach to implementing an application server herd a better solution.

To better understand how asyncio would work in practice, I wrote a prototype program that uses asyncio with an application server herd of five servers. Some of the servers do not communicate with each other and there is a pattern for which servers communicate with each other. Using a TCP connection, a client can send a server their location and the location is quickly shared with the rest of the servers using a flooding algorithm. Then another client can query about places nearby this client's location. The server will then send a HTTP request to the Google servers to query the Google Places API with the client's location that it has saved. The asyncio library only supports TCP and SSL protocols, so a different library, aiohttp will be used. After getting a response from the Google servers, the asyncio-based server will send the client back a JSON-formatted message with places nearby the requested client's location. Each server also logs its input, output, and when it connects/disconnects from another server.

I will briefly compare the overall approaches of Python's asyncio library and Node.js. Similarly to asyncio, Node.js also uses one thread and one process and uses cooperative multi-tasking. So asyncio and Node.js are based on the same idea, but they handle concurrency in different ways. For example, asyncio uses awaitables (coroutines, async Tasks, or Futures) while Node.js uses Promises.

This paper is organized as follows: Section 2 discusses asyncio's source code and documentation. Section 3 discusses the possible limitations of using Python with bigger applications when it comes to type checking, memory management, multi-threading, and other language-related issues. Section 4 describes the prototype I wrote and problems that I ran into while writing it. Section 5 is a brief comparison of asyncio and Node.js. Section 6 concludes this paper and gives a recommendation of whether asyncio would be an effective way to implement an application server herd.

2 Python's Asyncio Library Analysis

Asyncio provides a set of low-level and a set of high-level APIs. The high-level APIs include running coroutines, performing network IO and IPC, controlling subprocesses, distributing tasks via queues, and synchronizing concurrent code. While the low-level APIs include creating and managing event loops, implementing efficient protocols using transports, and bridging callback-based libraries and code with async/await syntax. In this paper, I will mostly talk about the high-level APIs that I used in my prototype and briefly discuss event loops.

2.1 Coroutines and Network IO and IPC

A coroutine is a function that can suspend its execution and give control to another coroutine. When a function is declared, if you put “async” in front of “def” then the function becomes a coroutine. The asyncio.run() function will need to be used to run the top-level entry point function since simply calling a coroutine will not schedule it to be executed. Then when you call other coroutines, you can put “await” before the function call to suspend the execution of the current coroutine until the awaited function is finished. Asyncio also gives you the ability to create Tasks which are used to schedule coroutines concurrently. Lastly, there are Futures which are low-level awaitable objects that represent an eventual result of an asynchronous operation.When a Future object is awaited it means that the coroutine will wait until the Future is resolved in some other place. However, the creation of Future objects is usually not needed at the application level.

Streams are high-level async/await-ready primitives to work with network connections. The asyncio.open_connection() function is used to establish a network connection and returns a pair of (reader, writer) objects. Then the asyncio.start_server() function starts a socket server and the first argument is a callback. It receives a (reader, writer) pair as two arguments. A callback is a subroutine function which is passed as an argument to be executed at some point in the future. In this case, the callback function is called whenever a new client connection is established. The reader object previously mentioned is an instance of the StreamReader class which provides APIs to read data from the IO stream. Similarly, the writer object is an instance of the StreamWriter class which provides APIs to write data to the IO stream. One of its coroutines is drain() which is a flow control method that waits until it is appropriate to resume writing to the stream.

2.2 The Event Loop

The event loop runs asynchronous tasks and callbacks, performs network IO operations, and runs subprocesses. Application developers will usually just use the high-level asyncio functions rather than using the loop object and its method. However, it is important to understand that the event loop is what is behind these high-level functions. The event loop is in charge of scheduling the execution of coroutines, opening network connections, creating network servers among other things.

Looking at the source code for asyncio, it can be seen that event loop is used implement all of the high-level APIs mentioned above. For example, the create_task() function, which schedules the execution of a coroutine object, gets the running event loop and uses the loop object’s method create_task() to schedule the execution of the coroutine object. So the high-level APIs are wrappers for low-level APIs that use the event loop to perform cooperative multi-tasking.

3 Possible Limitations of Python

Python’s asyncio library has lots of useful high-level APIs that allow us to write concurrent code. However, the limitations of the language itself must be considered, especially if one wants to write bigger applications. In this section, I will look at how Python’s implementation of type checking, memory management, and multithreading compares to Java’s implementation. I will also address if it is important to rely on asyncio features of Python 3.9 or later.

3.1 Static vs. Dynamic Type Checking

Python uses dynamic type checking meaning that the Python interpreter does type checking only as code runs and the type of a variable is allowed to change over its lifetime. While Java uses static type checking where type checking is performed when the program is compiled and variables are generally not allowed to change types except with casting. So type errors will not be found in a program until the line of code with the error runs when using Python, but Java will find any type errors at compile time. This could cause problems for bigger applications where type errors could go unnoticed if the lines of code where they occur never run. Then the application developers will have to go back and fix these type errors in the future if they ever cause problems.

Considering how type checking relates to implementing an application server herd, it seems that static type checking would be a better choice to avoid unnoticed type errors. This type of application can easily be implemented with the types of all objects known before runtime and use casting to change types as needed. So in this case, a Java-based approach to implementing an application server herd looks like a better option. However, type checking is only part of writing code for an application.

3.2 Memory Management

Python uses a memory manager to manage its private heap which contains all Python objects and data structures. The allocation of heap space for Python objects is performed by the Python memory manager and the user has no control of it.

4 Prototype using Python’s Asyncio Library

Using the asyncio library, I wrote a server program that can run five different servers. I created coroutines by adding async in front of function declarations and used the await keyword to suspend the execution of coroutines.

5 Comparison: Asyncio vs Node.js

Node.js and asyncio are both based on the same idea of concurrency and cooperative multi-tasking. However, Node.js

uses more low-level APIs. So an application programmer has to understand more about how everything works than when using the asyncio library.

6 Conclusion/Recommendation

Considering language-related issues with Python when it comes to bigger applications, my own experience writing a prototype for an application server herd, and a comparison to Node.js, I would recommend that Python's asyncio library be used to implement an application server herd.

7 References

1. Eggert, Paul. "Project. Proxy Herd with Asyncio." *UCLA Computer Science 131*, Paul Eggert, https://web.cs.ucla.edu/classes/fall21/cs131/hw/pr.html.
2. Hjelle, Geir Arne. "Python Type Checking (Guide)." *Real Python*, Real Python, 3 Aug. 2021, https://realpython.com/python-type-checking.
3. "Introduction to Node.js." *NodeJS*, OpenJS Foundation, https://nodejs.dev/learn.
4. Python. "Cpython/Lib/Asyncio" *GitHub*, https://github.com/python/cpython/tree/main/Lib/asyncio.
5. Python Software Foundation. "Python 3.10.0 Documentation." *3.10.0 Documentation*, https://docs.python.org/3/index.html.
6. Solomon, Brad. "Async IO in Python: A Complete Walkthrough." *Real Python*, Real Python, 19 June 2021, https://realpython.com/async-io-python.