Report on using Asyncio library to implement a proxy herd

Keyu Ji, *University of California, Los Angeles*

Abstract

This report is written for the asyncio library, evaluating its pros and cons in terms of being used to implement a proxy herd. This report compares the pros and cons of the library itself, the features of Python with those of Java, and asyncio library with Node.js. It concludes that asyncio is theoretically better and some experimentation should be done for a definite answer.

1 Intro

Wikipedia has been well known because of its deep and width knowledge database contributed by people all around the world. As a result, there would be a large number of requests sent to the Wikipedia server either to update articles or to retrieve Wikipedia entries. We are now assuming that due to the heavy load, the single application server is a central performance bottleneck, and we are attempting to utilize a different architecture called "application server herd", where a number of servers serve together and communicate with each other to update information. In this way, the heavy load would be divided into lighter workload and response time would be shorten.

I have been looking into the asyncio asynchronous networking library as a good candidate for this new architecture. I have done research and written a simple exercise with Google Places API to evaluate how well asyncio would suffice suffice for the purpose. First, I will introduce the pros and cons of the asyncio library by itself. Second, I will compare Python against Java regarding type checking, memory management and multithreading. Then I will compare the overall approach of asyncio to that of Node.js. Finally, I conclude that asyncio is theoretically better.

2 Python

Every language has its own unique features and strengths. Python in this exercise also has a number of useful features.

In this exercise specifically, the proxy server would query the Google Places API for location information and the API would return an Json object. Python has the Json package which nicely supports parsing Json object. Considering the popularity of using Json to transmit structured data between server and web application, the support for easy Json object parsing is really nice to have.

Json package mainly transforms Json object into a dictionary, and the `dict` data structure in Python makes implementation of server a lot easier. As a server which faces many clients, more often than not it needs to save the client's information for future use. Each client's information needs to match each client's ID, and `dict` provides a native solution for this. The hash table also keeps the searching time in O(1), which helps with the server's performance.

3 Asyncio

3.1 Advantages

asyncio nicely allows concurrency for Python. Asynchronous implementation allows the program to do other tasks while waiting for the result. In our case, the program would need to wait for messages from the client and Google Places API, and instead of wasting time waiting, the CPU can turn around and do other useful work. For a proxy herd facing heavy workload, asyncio surely wins over any synchronous implementation and is a convincing reason why asyncio is a good choice for our purpose.

Asyncio by default utilizes TCP connection, which is the reliable data transfer protocol on the transport layer. It ensures all packets are delivered and receiver would receive them in order. It suits the purpose of a proxy herd as packet loss is undesirable. This default configuration is convenient.

3.2 Disadvantages

One never gets things for free in life. There are a couple of disadvantages of asyncio as well.

As opposed to common synchronous implementation, asyncio introduces many new concepts such as event loop and coroutine, which adds more complexity to program and debugging. There have been error messages that I did not fully understand.

Asyncio was just introduced in Python 3.4 and is still a very new module under development and subject to changes in the future. It has gone through several updates with Python already. To use asyncio, it requires the server host to have Python3.4 or later installed, and the programmers would need to be prepare for any changes done to the asyncio library, although I personally believe the designers of Python would keep the changes minimal and compatible.

4 Comparison with Java

4.1 Type Checking

In terms of type checking, Python utilizes dynamic type checking while Java utilizes static type checking. Dynamic type checking allows better flexibility, but also introduces more possibilities of type errors. In my opinion, the proxy herd is not object-oriented and does not necessarily desire flexibility. In addition, if the proxy herd scales up and needs to run more complicated applications, there might be unexpected type errors that weren't detected during development and may be invoked during real work, which would immediately bring the server down. Though a down proxy is not a big deal to a proxy herd, but people will want to avoid that. On the other hand, static type checking is far more reliable since all the type errors will be found during compilation. The priority of the proxy herd is to be reliable when serving clients. In this regard, Java's static type checking is better.

4.2 Memory Management

Python uses a runtime stack for activation records, which are references pointing to corresponding objects in the heap. Each object in the heap has its own reference count, which is simply the number of references pointing to it. Python's garbage collector is implemented with the idea that if an object in the heap has zero reference count, it is no longer needed and can be cleaned away. On the contrast, in Java, heap contains all objects created in application, and the garbage collector does its job via so-called "mark-and-sweep". First, all objects in the heap has one additional bit so that the garbage collector can mark it and unmark it. The garbage collector first clears all mark bits of all objects in the heap. Then it looks for all roots and mark objects they point to. Afterwards, find all marked objects and mark what they point to, and repeat this until a fixed point is reached. Finally, it removes all unmarked objects. As the length of explanation of implementation suggests, Java's approach would take more CPU resources to execute. Given that the proxy herd will deal with large amount of data, the performance of garbage collector matters. Python's garbage collector is better for us.

4.3 Multithreading

The difference between the memory managements of Java and Python leads to a difference in their multithreading performance. Java's memory model gives Java a great support for multithreading, and there are even more packages to manage thread synchronization. On the other hand, Python's memory management is not thread safe [4], and to prevent race conditions, a global interpreter lock is enforced to protect access to objects. The existence of the this mutex significantly lowers the multithreading performance of Python, so in terms of multithreading per se, Java has an edge. However, if we have to choose one, asynchronous implementation will be our priority since more time would be wasted if it was synchronous. As a result, the advantage of Java in multithreading does not edge over that of asyncio in server implementation. Still, the even better approach is to have multithreading and asynchronous implementation together.

5 Comparison with Node.js

There are many similarities between asyncio and Node.js, which is a Javascript runtime environment designed to use Javascript on web servers. It helps manage computing resources, file systems, and web application security. "Node.js is one of a few server-side solutions based on the concept of event-driven programming, which allows creating highly scalable servers without threading. On top of that, Node supports multiple concurrent requests and operations via asynchronous calls and non-blocking I/O [3]." Below is a comparison of Node.js with asyncio for our purposes to build a proxy herd for Wikipedia.

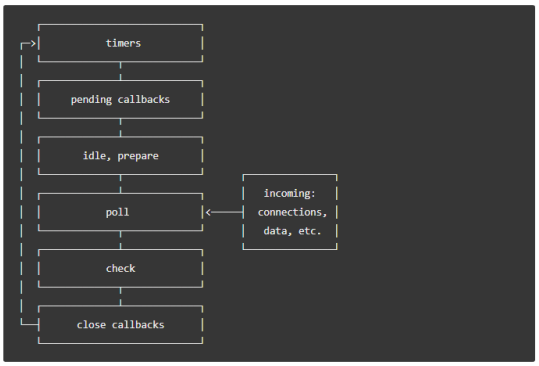
Node.js natively has an event-based architecture and non-blocking I/O. The event-driven architecture is based on asynchronous calls. The non-blocking I/O does not block program execution under I/O-heavy workloads. These two things together allow concurrency in programs written in Node.js. Also, by its nature, Asynchronous implementation is the default in Node.js. In asyncio, one has to use the keyword `async` to specify a "coroutine". In this regard, Node.js is easier to develop with.

asyncio is based on its own event loop and coroutine (cooperative routines). Notice that Node.js does not have the concept of a coroutine, and the event loop of asyncio is different from that of Node.js.

For asyncio, the event loop is simply a loop with tasks to run, and it runs them. It does not preempt a running routine but expects a coroutine to pause itself and allow the next task in the loop to run [2].

On the other side, the event loop of Node.js is like a group

of instructions that a supervisor makes repeatedly. The structure is shown below [1].



The most important element here is the timer, which suggests that Node.js does preempt a running routine when the timer reaches the configured threshold. However, without actual experiment,

Theoretically, in a network perspective, I think the the approach of asyncio is better for us since a coroutine yields when it wants and that makes the coroutine to finish as soon as possible. The timer of Node.js will not care about what the task is doing and cut it off immediately. Following on that, when a client makes a request, the server should process the request and send back a response as soon as possible. Given that the job of our proxy herd is rather simple, each coroutine will not take long to finish, so a voluntary scheduling algorithm can help the clients to get the response faster and improve the quality of service in this regard. However, I cannot say this with certainty without an actual experiment.

There are several advantages of Node.js by itself as well.

First, Node.js natively behaves like asyncio. Node.js has an event-based architecture and non-blocking I/O. The event-driven architecture is based on asynchronous calls. The non-blocking I/O does not block program execution under I/O-heavy workloads. These two things together allow concurrency in programs written in Node.js. Also, by its nature, Asynchronous implementation is the default in Node.js. In asyncio, one has to use the keyword `async` to specify a "coroutine". In this regard, Node.js is easier to develop with.

Second, Node.js has a standard library that provides modules that support high scalability. "Node clusters and workers are abstractions that can spawn additional Node.js processes depending on the workload of your web application [3]." Node.js can fully utilize the hardware resources of the system.

Lastly, Node.js and JavaScript can be utilized on both frontend and backend of a server, which eases the communication between them and development process.

Notice that although Node.js does not handle CPU intensive tasks well, since we are building a proxy herd merely for data storage and delivery, it should not be an issue.

6 Conclusions

In conclusion, asyncio library of Python is suitable for building a proxy herd that serves large amount of data storage and delivery. Programmers need time to learn the library itself and some concepts behind the implementations, but after that, it is relatively straightforward. Comparing Python with Java, Python has better memory management while Java has better reliability with static type checking. Comparing asyncio with Node.js, I personally think for us, the coroutine of asyncio is better than the timer of Node.js, but further experimentation is required for a definite conclusion.

References

[1] Node js Foundation. The node.js event loop, timers, and process.nexttick(), Apr 2019.

[2] Apoorv Garg. A simple introduction to python's asyncio, Jun 2018.

[3] Netguru. Node.js vs python comparison: Which solution to choose for your next project? | netguru blog on python, Jan 2018.

[4] Thomas Wouters. Globalinterpreterlock - python wiki, Aug 2017.