Project: Python Asyncio Feasibility

Abstract

In this report, the feasibility of implementing a complex server herd architecture using the Python Asyncio module for an application in the style of Wikimedia will be assessed. The ease of programming a basic prototype implementation will be discussed. We will consider how server inter-communication / flooding algorithms can be realized in such a module. The memory safety of the module will be discussed, alongside discussions of performance limitations of asyncio and python as a whole, as well as the importance of using newer python iterations for such implementations. Based on this analysis, recommendations regarding the use of this module in such a large-scale application will be made.

Introduction to Asyncio

Asyncio, at its core, is an implementation of concurrency into Python. It uses the concept of coroutines, that is, procedures which can be suspended and do not necessarily need to be completed before entering other routines. Concurrency is not to be confused with parallelism: as Asyncio implementations are single threaded, these coroutines are not running at the same time, even if they appear to. This implementation of concurrency, as the name implies, lends itself to the possibility of Asynchronous IO in python. Asynchronous IO is a concept where multiple IO activities can execute in non-sequential order. This allows for time-consuming activities to not block executing other elements of the program. Crucially, this kind of programming is needed for event driven programming, where some event may happen at any time that triggers execution of code. This is a core concept for the implementation of networked applications that send and receive messages, such as a proxy herd.

Asyncio is implemented with an event loop, which directs how execution of the tasks will occur. Functions that are to be made into coroutines must be marked asynchronous, and when called they must be awaited. The concept of networked applications is closely linked to Asynchronous IO, as Asyncio allows programs to remain useful during the downtime of waiting for responses over the internet, and it allows for the handling of multiple requests at once. Because of this, asynchio has a built-in routine that starts and handles TCP-capable servers.

Ease of Prototype Implementation

Asyncio programming presents a small learning curve for programmers who are unfamiliar with the principles of concurrent programming. The concept of the event loop are conceptually difficult for those accustomed to sequential or standard threaded programming. There was a plethora of documentation and guides available on the internet however which help ease this learning curve.

The biggest challenge I faced when implementing the prototype was the difficulties regarding implementing networking and starting a server. While this largely arose due to my unfamiliarity with writing such programs, I did find that the server-specific documentation for asyncio to be lackluster. This is likely due to the fact that asyncio is mainly seen as a module for the implementation of concurrency, and networking is just one of many uses for the module. Despite this, some guides are still available online, and when compared to other Asynchronous IO packages in lower-level languages, the learning curve was likely much less significant.

Once the basic framework was set up for sending and receiving messages over TCP, implementation was quite simple. Python features such as implicit types and simple casting made writing otherwise complicated code quite efficient.

To optimize the readability of the code and to capitalize on the features of asyncio, I made the decision to break the features up into many coroutines. There is a coroutine for handling IAMAT messages, which does some simple computation and text manipulation to store the client location in a global dictionary and return a response. This coroutine also calls upon a routine which propagates the message across the servers. As this flooding algorithm is one of the main focuses of this project, I have decided to include it in its own section. A coroutine was also written for handling WHATSAT requests,

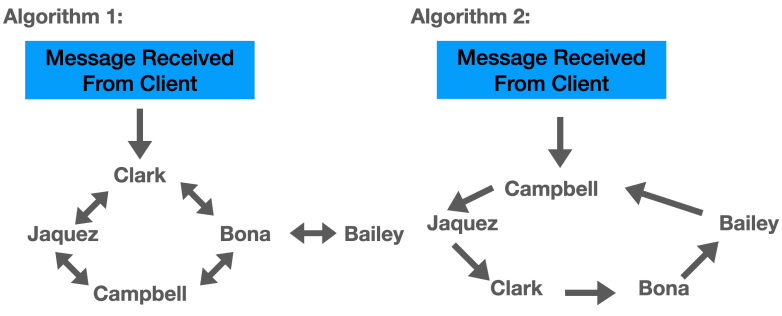
which calls upon coroutines for sending / receiving API requests from google places, as well as a coroutine which helps format the returned JSON message according to the specifications.

What greatly simplified this implementation was the large amount of reliable and useful modules built into the python language. It is difficult to find a language that has such a wide variety of packages freely available, and full-scale implementations of such a program would likely greatly benefit from access to these modules.

While learning the basics of asyncio and networked applications was a challenge, this learning curve was quickly overcome and the module proved quite simple and efficient to write useful code with.

Asyncio Server Herds Recommendations

Asyncio does not have a framework that is optimized specifically to run server herds, however implementing a flooding algorithm through regular TCP communication was relatively simple. With that being said, while not standard across implementations of asyncio, this module would benefit from having a framework to easily link servers for more hassle-free inter-server communication. This is quite self-implementable however, as it would be relatively simple to write a custom function that implements some protocol for communicating between servers.



As demonstrated in the diagrams above, there are two major, simple algorithms that could be implemented to solve the issue of propagation across the network.

In the first algorithm, when a server receives a message, it attempts propagates it to every server that it is able to communicate to. Then, the receiving server also propagates it to every server that it can communicate to. As each server keeps a log the IAMAT messages it receives, before propagating a message it checks if it had received an identical message previously. If it had, then it can safely assume it had already propagated this message and will not send it across the network.

A major drawback in implementing this first algorithm in a Wikimedia-style server herd would be, due to the naive approach of sending and receiving messages that does not consider which servers the message had previously been sent to, many unnecessary requests would be sent and received by servers, which could get in the way of running useful code (although this issue greatly mitigated by using Asynchronous IO). This problem would be especially prevalent in applications where all or nearly all servers in the herd have the capability to all talk to each other. This could be mitigated by adding logic to differentiate client-server and server-server communications, and then sending over a data structure that keeps track of which servers had already received the messages and which still needed to be reached. This approach still has some drawbacks however, as this data structure could be quite large in implementations with many servers and the algorithm could become quite complicated.

In the second algorithm, the messages are sent in a cyclic fashion. The advantage of this is that sending a message is always productive, and the only case where a server receives a message that it had already gotten would be when the cycle returns to the original receiving server, which would know to stop the message propagation from there. This would be ideal in cases where inter-server communication is greatly limited. This implementation has major concerns in the case of redundancy, in the case of a server going down the cycle would not be able to be completed. Additionally, this would be a relatively slow implementation, as it is O(N), each server would have to wait for all the previous servers to receive messages.

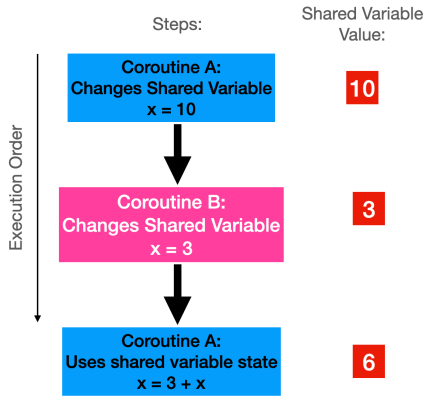
In my opinion, the best approach for a Wikimedia style production herd would take a hybrid approach between both algorithms. For use cases where speed of propagation across the network does not matter, such as adjusting

the like count of a Wikimedia article, then the second algorithm would be ideal as it would alleviate network traffic. In the rare case that a server is down, a backup strategy could be implemented where alternative cycles without the disrupted server are attempted, and if that fails, it could fall-back on sending messages through algorithm one. When speed is necessary, then algorithm one should be implemented, potentially with the recommended message-tracking adjustments.

Asyncio Memory Safety

Asyncio implementations would be a good candidate for such a server-herd given their relative memory safety.

As Asynchio is single-threaded, it would not be possible for some shared state to be updated at exact the same time that some other thread updates the same shared state, thus most race conditions are not possible.



As demonstrated in the example diagram, it is important to note that concurrency bugs can arise from shared state accesses in asyncio programs regardless of single-threading. As coroutine instructions are interleaved, one coroutine can modify the shared state, and then access the value again in subsequent accesses expecting no changes, only for some interleaved instructions to have unknowingly modified the shared access state.

It would be extremely important to take these kinds of concurrency problems into account, as when the amount of requests greatly increases in full-scale servers, the number of instances of coroutines being ran would grow in tandem and the chances of these bugs manifesting would be much greater. These concurrency bugs can be easily avoidable by writing smart code. They would also likely manifest in other programming languages implementations of Asynchronous IO, as this is an inherit problem that comes with concurrency. Because of this, Asyncio remains a relatively safe and reliable choice for writing production server herds.

Performance Implications

Performance would be the greatest limiting factor in the implementation of a full-scale Wikimedia server herd.

In terms of performance benefits from using Asyncio, the implications of concurrency are that the machine will not be held up by any server coroutine, allowing it to handle many requests simultaneously. This would allow the server to sustain and respond to a large number of connections at once.

Python is notoriously slow, with an analysis conducted by FreeCodeCamp comparing the speeds of Python and C++ for various identical algorithms finding that C++ beat Python in every single case. Before delving into the performance of the Asyncio module itself, it's important to consider that the use of Python could be a potential bottleneck. This may not be as big of an issue as it seems, as these benchmarks were conducted doing large-scale data analysis, which is quite different from the use case of Asynchronous IO implementations which generally would not employ such intensive algorithms for simple sending and receiving.

The single greatest reason that Asyncio would not be able to scale for such a large-scale application is the lack of multithreading in its implementation. While Asyncio may be able to run concurrent code, all of this code is still run on a single core. This greatly increases the safety and reliability of applications by avoiding race conditions, however this has huge performance implications at scale. Real world servers would likely have many cores, which would be crucial to utilize when handling a large number of requests every second. It would be a considerable waste of computational resources, and a sunk cost when considering the cost of the multi-core CPU, to write a program for such a machine that only runs on a single core. Single-threading in such a

complex application would mean that it would be simply unable to keep up.

All is not lost however, and there would still be some valid uses for asyncio. Asyncio would still be a great option for a server-herd in less request-intensive uses. Additionally, the other cores of the CPU are not necessarily lost and would benefit from being utilized by other critical software (and operating system uses). Asyncio, with smart and intentional implementation, could work effectively in this scenario. An approach could be implemented where intensive tasks are offloaded to a separate core, freeing the asyncio core up for only handling simple requests and responses.

Reliance on Newer Features

Python 3.9 marked the release of several important changes to the asyncio module.

The *reuse_address* parameter was removed from the module, which had significant security vulnerabilities. Using older versions of asyncio would leave this security issue open, which would be an important consideration given the scale of this application.

A new coroutine *to_thread* was added to the module, which allows the offloading of execution of routines to be ran in a blocked manner on a separate thread. As mentioned in the "*Performance Implications*" section of this report, employing offloading to separate threads would be of crucial importance for this application in order to free up computational resources for handling sending / receiving requests. While this was technically possible pre-Python 3.9, it was lower-level and thus would have caused some programming challenges in otherwise simple implementations.

Outside of these, Python 3.9 also introduced some small optimizations that could help make the module more suitable for production-level code. This includes the introduction of a new coroutine *shutdown_default_executor* that allows for a cleaner, more reliable shutdown. Additionally, some small changes such as improvements to the wait_for routine and SSL socket type checking would help improve reliability.

While pre-Python 3.9 asyncio is perfectly usable for such an implementation, using newer versions would allow for a more reliable, safer, and more optimized experience.

A Note on Type Checking

Python is generally not explicitly typed, which greatly increases convenience on the programmer, however it comes at a cost of program reliability.

To overcome the lack of explicit types in Python, dynamic typing is used, meaning that type adherence is checked at runtime. This poses the issue of how Python determines the types in the first place, which it does through a principle called duck typing, where types are implicitly determined by examining how the data is used. Due to these principles, changing object types in Python is quite simple, with simple built-in functions allowing casting to different types.

These unique type principles do leave room for some error, the most common of which being programmer error. Frequently switching between different types may lead to some confusion, causing some methods incompatible to the current type incorrectly being run. Most notably however, the lack of static type checking means that errors where types are not adhered to would likely not be caught until runtime. This could decrease program reliability.

While it is important to consider the impacts that dynamic type-checking would have, it is likely not as large of a problem as it seems. The python type system is relatively standardized, and overloaded methods can be used amongst similar types. Python also recently introduced type hints, which will allow the interpreter to do some light static type checking which help catch blatant type violations.

Node.js vs Asyncio Comparison

Node.js is quite similar in most ways to asyncio, and would likely be just as good of an avenue to implementing this application. Node.js is also a method of doing Asynchronous IO, and uses the concept of an event loop. Node.js uses the concept of async and await, making the functionality and implementation strikingly similar to Python Asyncio. Implementation-wise there would likely be little difference between both platforms.

Both implementations are similar in their performance, with each being single-threaded with some options for increasing threading. Node.js seems to have some more performance capabilities however, with a comprehensive

performance API as well as better multithreading options. Additionally, it is more commonly used for real-world high performance applications.

The biggest difference between them is their community and software support. Node.js is hugely popular and widely used for sever applications, so there would likely be vastly more resources and programmers familiar with the framework compared to Python Asyncio.

Final Recommendations

I would recommend not using the asyncio module for the implementation of the wikimedia style server herd.

Asyncio is extremely well-suited for some small to medium scale applications, offering ease of implementation and efficiency in writing code. It lacks, however, in its ability scale.

The primary reason why I would not recommend the use of the python Asyncio module is due to the single-threaded nature of the concurrency that is implemented. While most Asynchronous IO options among programming languages are also single-threaded by default, other languages seem to offer better options for overcoming this performance detriment.

I would also not recommend the use Python Asyncio in this application due to the lack of community guides and support available for the module. In my research I found that other frameworks for implementing concurrency were considerably more well-established in the space, such as Node.js. Most developers at the Wikimedia company would likely already be familiar with the Node.js framework and be able to implement the server herd with relative ease. Using this lesser-used module in such a complex, high-performance application would be risky and there may not be sufficient documentation, and it would be more likely to run across novel bugs in the module compared to Node.js.

There are also some other concerns that I have with the use of Asyncio. Having a server go down could be disastrous, which means it would be important to catch all possible errors at compile time. Due to the dynamic typing in Python, this is not possible. Another concern is the significant changes in newer iterations of the asyncio module would decrease backwards compatibility with code written for older versions.

In conclusion, Asyncio is a great, powerful module for Python that empowers people to easily implement otherwise complex applications. However, it is not nearly well-established enough, nor performant enough, to justify using in such a large and important application. Instead, other popular Asynchronous IO frameworks should be investigated, such as Node.js. The Asyncio module, and the python language, is constantly evolving and improving. Thus, this topic should be revisited further down the line, as it is likely that asyncio will soon become an exceptional alternative to other Asynchronous IO frameworks.

Sources

Behery, Anthony. “Python vs C++ Time Complexity Analysis.” *FreeCodeCamp.org*, 1 Mar. 2023, www.freecodecamp.org/news/python-vs-c-plus-plus-time-complexity-analysis/.

Langa, Łukasz. “What’s New in Python 3.9.” *Python Documentation*, 5 Oct. 2020, docs.python.org/3/whatsnew/3.9.html#asyncio.

Notna, Andrei. “Intro to Async Concurrency in Python and Node.js.” *Medium*, 12 Mar. 2019, medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36.

“Python Event-Driven Programming.” *GeeksforGeeks*, 27 Mar. 2024, www.geeksforgeeks.org/python-event-driven-programming/.

Python, Real. “Async IO in Python: A Complete Walkthrough – Real Python.” *Realpython.com*, realpython.com/async-io-python/.

---. “Python Type Checking (Guide) – Real Python.” *Realpython.com*, realpython.com/python-type-checking/.

Singh, Utkarsh. “Advanced Guide to Asyncio, Threading, and Multiprocessing in Python.” *Medium*, 21 Dec. 2023, connectwithutkarshsingh.medium.com/advanced-guide-to-asyncio-threading-and-multiprocessing-in-python-c4dc50971d24.

Sufiyan, Taha. “What Is Node.js: A Comprehensive Guide.” *Simplilearn.com*, www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs#:~:text=Developers.