Python Project: Asyncio Proxy Herd

Alex Du
University of California, Los Angeles

ABSTRACT

The current Wikimedia platform used by Wikipedia is not suitable for a new Wikimedia-style service designed for news due to several reasons: article updates will happen too often, various protocols other than HTTP or HTTPS will be required, and clients will be more mobile. The PHP and JavaScript application server used by Wikimedia looks like it will be a bottleneck, as it will be difficult for the application to add newer servers, and the response time will be slow because the Wikimedia application server is a bottleneck on cores. This paper will focus on a Google Places proxy prototype built on Python's asyncio module that has an architecture that could potentially replace Wikimedia's traditional one and will also evaluate the pros and cons of asyncio.

I. INTRODUCTION

In this research paper, we explore the potential of using Python's asyncio asynchronous networking library as a replacement for part or all of the Wikimedia platform for a new type of service designed for news. To overcome the limitations previously outlined, we propose an "application server herd" architecture, where servers communicate directly with each other, rather than relying on a central server, as did Wikimedia. In this model, the database server is still used for infrequently accessed data, while the application servers handle rapidly evolving data, such as GPS-based locations and ephemeral video data (Eggert).

To evaluate the effectiveness of using Python's asyncio library for our purposes, we conduct a thorough examination of the asyncio source code and documentation. We also develop a prototype application 'sever.py' consisting of five servers that communicate bidirectionally with each other, simulating the behavior of mobile devices. Clients can send their location information to a server, which is then propagated to other servers using a simple flooding algorithm.

Our evaluation considers factors such as ease of development, maintainability, and reliability of asyncio. We also investigate the performance of the prototype in terms of response time and scalability and compare it to that of the Wikimedia platform.

Overall, our research provides valuable insights into the feasibility of using asyncio for large-scale, real-time web services, and demonstrates the potential of application server herd architecture as a way to improve performance and scalability.

II. BUILDING THE PROTOTYPE

The server design is based on a simple messaging protocol using TCP connections. There are five servers in total, named Bailey, Bona, Campbell, Clark, Jaquez, each given different ports and IP addresses, and they communicate with each other using TCP (Eggert).

The servers have a shared memory containing a database of client locations, which is updated as new client locations are received. Each contains information about clients' ID, latitude and longitude, timestamp, and the server that originally received the location. There are three types of commands that clients can send to the server: 'IAMAT', 'AT', and 'WHATSAT.'

The 'IAMAT' command is used to report a client's current location to the server: it sends a message containing its ID, latitude and longitude, and timestamp. The server records this information and sends a response back to the client confirming it has processed the information.

The 'AT' command is used by servers to communicate with each other (during propagation) and update their databases. When a server receives an IAMAT command from a client, it sends an AT command to all other servers in the network after it has processed the information on its own. Each server that receives an AT command updates its own database with the client's location information, but does not send another AT command to other servers.

The 'WHATSAT' command takes three arguments: a client ID, which must already be in our database, populated by a previous "IAM" command, a radius, and a maximum number of results to return. When the server receives a WHATSAT command, the server retrieves the latitude and longitude of the location specified in the corresponding "IAMAT" command, then uses the Google Places API to retrieve information about places in the vicinity of that location, up to the specified radius and maximum number of results, and returns Google's API GET result formatted as a JSON object.

In terms of the TCP connections between servers, each server opens a listening socket and waits for incoming connections. When a server receives a connection request from another server, it accepts the connection and creates a new thread to handle the communication. Each thread runs a loop that continuously listens for incoming messages from the connected servers.

Thus, the server design is based on a simple messaging protocol using TCP connections. Clients can send commands to the server, and servers communicate with each other and update their databases.

III. ANALYSIS OF ASYNCIO

We will evaluate asyncio in terms of factors such as its ease of development, maintainability, and reliability of asyncio. We will compare these with Node.js.

In terms of development, asyncio and Node.js use completely different approaches: Node.js uses an event loop that runs non-blocking I/O operations, while asyncio is based on Python's asynchronous coroutines and uses an event loop to manage task scheduling (Notna). More specifically, Node.js has a simpler and more straightforward API, making it much easier to learn. However, this has a side effect that it is less flexible and requires more code to handle certain tasks. On the other hand, asyncio has a steeper learning curve, since it utilizes coroutines with the event loop. However, the code is more concise and can take advantage of Python's language features; asyncio can be integrated with other Python libraries and tools, making it suitable for more complex projects.

In terms of maintainability, asyncio can be considered an excellent choice for developing asynchronous Python applications, especially when compared to other options such as Node.js. The structured way to organize code into coroutines, tasks, and event loops will naturally modularize the code, which makes it easier to read and debug (Notna). Moreover, there are built-in debugging tools available (our server prototype logs events). Finally, it is likely to remain compatible with future versions of Python; in the case it doesn't, standard Python testing frameworks and extensive asyncio documentation helps ensure changes can be easily made to the code that fix new bugs.

IV. ANALYSIS OF PYTHON

We will evaluate Python's type checking, memory management, and multithreading, and compare them to a Java-based approach to this problem.

In terms of type checking, Python is dynamically typed, meaning that the type of a variable is determined at runtime, which can lead to more runtime errors. On the other hand, Java is statically typed, which means that the type of a variable must be known at compile-time. This makes Java code less error-prone, but harder to read.

In terms of memory management, Python uses dynamic memory management, which means that memory does not need to be allocated explicitly, and the garbage collector takes care of deallocating memory that is no longer used (Real). This approach makes it easier to write code and can be beneficial for rapid prototyping or for programs that have a short lifespan. However, it can also lead to performance issues in long-running applications, as the garbage collector may introduce overhead.

While memory management is also automatic in Java, its garbage collector periodically frees memory that is no longer used. Java also has more purposeful allocation and deallocation (Java), which can be beneficial when memory being used is large, since programmers have much greater control over it. Therefore, while Java's memory management can be considered more robust and efficient than its counterpart, Python's is more convenient when implemented.

In terms of multithreading, Python includes a Global Interpreter Lock that prevents threads from executing Python bytecode in parallel (Notna). Consequently, Python is not as appropriate for CPU-bound multithreaded applications. The asyncio library provides an alternative approach to concurrency that is based on cooperative multitasking utilizing coroutines and can be used to write I/O-bound applications, since these are much more likely to be bound to single threads; once again, there can inherently be no multithreading here. Java, on the other hand, has extensive built-in support for multithreading (as seen in Homework 3) and provides a rich set of tools and libraries for writing these sorts of applications.

In terms of the prototype described in this report, both Python and Java could be used to implement the server application. Python's dynamic typing and automatic memory management could make it easier and faster to write code, while Java's static typing and more efficient garbage collector could make it more reliable and efficient, with its precision in memory management. If the application is truly I/O bound, then Python asyncio's concurrency is satisfactory; however, if the server needs to handle a high volume of concurrent connections that need to be propagated to (i.e. a large number of mobile clients), Python's global interpreter lock could limit its scalability, while Java's support for multithreading would make it the superior choice.

IV. ADDRESSING OTHER LANGUAGE ISSUES

We now analyze how easy it is to write asyncio-based programs that exploit server herds and its performance implications, as well as the importance of relying on asyncio features in Python 3.9 or later, including asyncio.run and python3 -m asyncio.

Writing asyncio-based programs that run and exploit server herds can be relatively easy or difficult depending on your experience with asynchronous programming and the complexity of the project you are working on. Prior familiarity with asynchronous programming concepts such as event loops, coroutines, and futures made it relatively easy to understand asyncio-based programs that run and exploit server herds. However, it may be more challenging to get started with understanding how to structure the code correctly if these concepts are foreign

As for the performance implications of using asyncio, it can improve the performance of your program by allowing for non-blocking operations. This means that if the program is I/O bound, then asyncio's ability to perform operations such as reading and writing to files or sockets without blocking the event loop can result in a significant performance increase. It is important to note, however, that managing coroutines and the event loop can introduce some additional overhead.

Regarding the importance of asyncio features of Python 3.9 or later, such as asyncio.run and python3 -m asyncio, it depends on specific needs and requirements, but in general newer Python versions are slightly faster and more reliable than previous versions, with a lot of minor improvements in areas like memory management, or in the case of asyncio, minor improvements to the efficiency of the event loops. Python 3.7 introduced asyncio.run (Runners), which helps simplify the code a lot by handling the set-up and clean-up of the event loop for the programmer, which grants noticeable improvements in both readability and simplicity. Using python3 -m asyncio invokes a read-eval-print-loop that is similar to a command prompt, so it can definitely help with experimentation and familiarization with the asyncio library. Both asyncio.run and python3 -m asyncio are features introduced in recent Python versions that improve the convenience of programming but are not absolutely necessary for the successful implementation of our prototype server.

V. CONCLUSION

The results of our research show that Python's asyncio library can be an effective tool for implementing the application herd. Asyncio is a powerful and flexible framework for building concurrent, event-driven, and high-performance applications, making it well-suited for an application server herd that requires speedy communication between servers. Additionally, it appropriately supports TCP and UDP protocols, and by nature, its asynchronous capabilities enable it to handle a high volume of I/O. Finally, it is extremely simple to integrate asyncio with other Python libraries or frameworks.

However, there are also some potential drawbacks to using asyncio. One is that it can be more complex to write and debug than traditional synchronous code. Another is that it may not be the best choice when there are more rigorous performance requirements from the application, due to its inherent incompatibilities with multithreading.

VI. WORKS CITED

Eggert, Paul. Project. Proxy Herd with Asyncio, https://web.cs.ucla.edu/classes/winter23/cs131/hw/pr.html.

“Java Memory Management.” GeeksforGeeks, GeeksforGeeks, 14 Dec. 2018, https://www.geeksforgeeks.org/java-memory-management/.

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

Real Python. “Memory Management in Python.” Real Python, Real Python, 10 Apr. 2021, https://realpython.com/python-memory-management/.

“Runners.” Python Documentation, https://docs.python.org/3/library/asyncio-runner.html.