CS 131 Project: Proxy herd with asyncio

Tingyu Gong
University of California, Los Angeles

Abstract

The purpose of this project is to research Python's *asyncio* asynchronous I/O networking library as a suitable candidate framework for replacing the Wikimedia platform with an "application server herd." This paper will discuss the prototype Python proxy code for the Google Places API, explore the pros and cons of using *asyncio* in practice, examine Python's type checking, memory management, and multithreading in comparison with Java, and compare the overall approach of *asyncio* with Node.js, to determine whether *asyncio* is an effective and reliable choice in implementing the application server herd.

1. Introduction

Wikimedia is a server platform written in PHP+Javascript that websites like Wikipedia are based on. It uses the Debian GNU/Linux, Apache web server, Memcached cache, and MariaDB database. Currently, we are looking to build a new Wikimedia-style service designed for news, where updates to articles are more often, access is required via various protocols (not just HTTP or HTTPS), and clients will be more mobile. The current PHP+JavaScript is unsuitable for this new service, as it will be a bottleneck. Our team has been tasked with looking into an alternative architecture, called the application server herd.

1.1 The Application Server Herd

An application server herd is an architecture where multiple application servers communicate directly to each other via core database and caches. This type of architecture is useful for fetching rapidly-evolving data, such as GPS-based locations. For this project, we develop a server herd in Python, where users can send a message to the servers for information about places near their location. The server responds by fetching data from the Google Places API. Since the data is rapidly-evolving, storing this location info on the database will be too slow due to bottleneck on the core clusters. Instead, it's more efficient to have servers talk to each other and propagate the user location to the other servers.

2. The Prototype Server Herd

This application server herd is implemented with Python's *asyncio* networking library. The server head is first started using *asyncio.start_server()*, which creates a TCP server. Users can send their GPS locations to the server herd, which fetches nearby places with Google Places API, and responds to the user with the data. To query the Google Places API, an HTTP request for data must be sent to the Google servers. As *asyncio* only supports TCP and SSL protocols, the server herd manually creates and sends an HTTP GET using the *aiohttp* library, which is a asynchronous HTTP client/server for *asyncio*.

2.1 Servers

First, let's introduce the servers. In this prototype, we have set up five servers each with their own ports. Their server IDs are Bailey, Bona, Campbell, Clark, and Jaquez. They communicate bidirectionally with each other in the following pattern:

1. Clark talks with Jaquez and Bona.
2. Campbell talks with everyone else but Clark.
3. Bona talks with Bailey.

2.1 Messages

Each server accepts Transmission Control Protocol (TCP) connections from clients that emulate mobile devices with IP addresses and DNS names. The servers must be able to handle three specific messages. They are as follows:

1. Clients can send its location to the server with an IAMAT message. The message is in this format:

IAMAT *clientID latitude-longitude time*

2. The server should respond to the client with an AT message and propage this message to neighboring servers. Client messages are saved in history to prevent infinite propagation. The message is in this format:

AT *serverID timeDifference clientID latitude-longitude time*

3. Clients can query the server for places near the client's location with a WHATSAT message. The server parses this message to send a GET request to the Google Places API, which returns the nearby places. The message is in this format:

WHATSAT *clientID radius bound*

The *clientID* identifies the client that is sending a message. The user's location is in the form of *latitude* and *longitude* coordinates. The *time* is expressed as POSIX time and represents the time the client sent the message. The *serverID* identifies the server that received the message from the client. The *timeDifference* is the time difference between when the server received the request and when the client sent the request. The *radius* is the how far in kilometers the client wants to search for places. The *bound* is the upper bound on the amount of information to receive within the radius.

If the client sends a message that is not one of the above, then it's considered an *invalidMessage*. The server responds to the invalid message in this format.

? *invalidMessage*

3. Pros and Cons of *asyncio*

asyncio (asynchronous I/O) is a library in Python that allows for writing concurrent asynchronous code using *async/await*. Asynchronous I/O refers to the process of executing other processes while waiting for an I/O process to be completed. It's often used for I/O-bound and high-level structured network code [1]. Defining a method with *async def function()* creates a coroutine, which is a function whose execution can pause in order to execute another coroutine. Coroutines are "awaitable," meaning it can be used to *await* until another task is completed. Event loops are another part of the *asyncio* model, and they can schedule and manage multiple tasks at a time. Tasks can be deliberately created with *asyncio.create_task()* [1]. These features are particularly important for managing I/O in TCP servers, which constantly send and receive messages.

3.1 Advantages

Implementing the server head with *asyncio* has multiple advantages due to asynchronous programming, which can allow the project to efficiently manage network messages.

When establishing connections between servers in the network, *asyncio* can allow for connections to be made as coroutines. In doing so, servers can communicate with each other through a flooding algorithm that propagates messages all at once asynchronously. For a large network, establishing connections asynchronously will be much more efficient than attempting to create connections one by one. In addition, with *asyncio*, the server does not need to waste CPU time by waiting for client queries or writing to clients. While waiting for a coroutine to finish, the server can switch to another process so that multiple tasks can be completed at once. This allows for servers to both continuously receive client requests and process messages, and thus boosting performance of the server head. When the server requests the Google Places API for data, *asyncio*, using the *aiohttp* library, can also define the API request as a coroutine. Instead of pausing the progress of the entire server head to wait for the GET request to return fetched data from the Google Places API, the server can switch to another process for better efficiency. As a result, *asyncio* provides various asynchronous programming features to run multiple tasks that can maximize the performance of network writes and reads.

3.2 Disadvantages

Issues can arise in *asyncio* code that relies on the ordering of tasks. With *asyncio*, order of tasks is based on order of completion, not order of execution. There is no guarantee that the event loop will execute tasks in order. This can cause issues in the project herd as IAMAT messages are expected to be processed first so that client location is stored in data. WHATSAT, which utilizes the stored location for queries, may be executed prior to IAMAT. This can result in unwanted errors in message handling [2]. In addition, although *asyncio* handles I/O-bound tasks efficiently, it does not inherently improve CPU-bound performance due to its single-threaded nature. Tasks that require heavy computation might cause bottlenecks and will need to be offloaded to separate threads or processes [3]. While asyncio is well-supported in Python 3.11.2, integrating it with libraries or existing code that are not designed for asynchronous execution can also be challenging. This may require significant code rework or use of alternative libraries [2].

3.3 Conclusion on *asyncio*

Learning asynchronous programming may be challenging due to unfamiliar tools and applications. However, the Python documentation for *asyncio* is well-written and thorough, and contains detailed examples. As such, writing *asyncio*-based programs is not an overwhelmingly challenging task [1]. In terms of performance, due to *asyncio*'s ability to handle and make connections all at once through asynchronous operations, efficiency is relatively high. Unless a program requires multithreading or CPU-intensive tasks, as *asyncio* is single-threaded, there are no major negative implications on performance [3]. *asyncio*'s optimality is based on its ability to have better control and complete I/O tasks

over threading. While the newer features of *asyncio* in Python 3.9 and later can make asynchronous programming easier and more efficient, it's still possible to use *asyncio* effectively in older versions of Python. This server head project, specifically, does not heavily rely on Python 3.9. For example, *asyncio.run()* is a function that offers a simple way to run a coroutine [1]. It has since been updated to run the newer coroutine *shutdown_default_executor()*, which schedules a shutdown for the default executor [1]. This update did not impact the core functionality of the server head prototype. The command line prompt *python3 -m asyncio*, which runs asyncio programs in an *asyncio* REPL in the terminal, also does not affect the server head program. The Python language does not drastically change with each new update. Thus, older versions of Python are still very viable.

To conclude, *asyncio* is suitable for this application server head, specifically for improving server/client messaging efficiency. The advantages of using *asyncio* far outweighs the disadvantages, which can largely be solved with structured code. *asyncio* is a powerful tool for writing concurrent code and very beneficial to projects dealing with communication in large networks.

4. Python vs. Java

We will now compare type checking, memory management, and multithreading between Python and Java to determine which is the more suitable language for building the application server head.

4.1 Type Checking

Python is a dynamically typed, interpreted language, where variable types are performed at runtime. This means data variables can hold different data types throughout the program, offering flexibility in coding. However, this requires careful coding practices in order to avoid errors, as Python can compile code even when issues are present. On the other hand, Java is a statically typed, compiled language, meaning variables must be declared before use. This enforces stricter type checking, which can catch errors early during development. However, this also means that all errors must be resolved before compilation can be continued [4].

Between the two, Python is a much more development-friendly, albeit error-prone language, due to its flexibility. While Java's typing system ensures programs can be run without errors, development with Java can be frustrating due to its strict compiler. In the end, deciding between the two languages is dependent on developer preference between flexibility or control.

4.2 Memory Management

Both Python and Java handle memory management with different takes on garbage collection [4]. Garbage collection automatically frees memory of objects that are no longer used without the need for assistance by the developer. This can prevent vulnerabilities like memory leaks and make managing memory easier. For Python, the virtual machine uses its garbage collector to free up memory when not in use. Python also has reference counting, which counts the number of times an object is referenced by other objects. When the number of references drops to zero, the object is automatically deallocated. Interfaces like this allow for high-level and automatic memory management in Python. However, issues with reference counting arise when two objects reference each other, creating circular references. This causes the reference count to never drop to zero and can lead to memory leaks and unwanted overheads [5]. For Java, its Garbage Collector (GC) is a component of the Java Virtual Machine (JVM) responsible for managing memory. The GC utilizes a mark-and-sweep algorithm that traverses through objects and marks objects that are "reachable," which means that the program maintains a reference to it. During the sweep phase, the GC iterates over objects on the heap and frees the memory of objects that are not marked. Though this algorithm is reliable, overheads may be introduced as the JVM has to track and iterate through all objects. This can lead to significant costs for larger programs that require immense memory usage [6]. As a result, choosing between Python or Java again depends on the developer and type of program being written.

4.3 Multithreading

Multi-threaded programs do not work well with Python. Since Python uses the Global Interpreter Lock (GIL) for reference counting, code that utilizes multiple threads suffers from obstruction as the GIL locks the interpreter when updating reference counts to avoid data races. Multithreaded applications in Python do not take advantage of the number of CPU cores the system may have for doing parallel tasks. However, workarounds exist that can solve Python's multithreading problems with the GIL. Using multiprocessing, a built-in Python module, additional interpreters can be created for implementing different processes at once [7]. Running multiple interpreters like this can be resource hungry, though. Java, however, works very well with multithreaded processes and offers various synchronization tools. The JVM manages threads, provides classes for

implementing threads, and handles memory for multithreaded applications, enabling developers to create efficient and reliable multithreaded programs.

5. *asyncio* vs. Node.js

asyncio and Node.js share many similarities for handling asynchronous programming, with a few key differences. While *asyncio* is a Python library, Node.js is a JavaScript runtime environment that can execute Javascript code outside of a web browser. JavaScript, a single-threaded language, struggles with parallelism, just like Python. Node.js is a possible solution to handling JavaScript's multithreading. Both tools utilize a single-threaded event loop model. While *asyncio*'s await/async uses coroutines, tasks, and awaitable objects to handle I/O-bound tasks, Node.js's await/async is built in on top of a similar mechanism called promises [8]. Promises are used for handling asynchronous operations by having a pending, resolved, and rejected state when requesting data. When data is received incorrectly, promises can invoke callback functions to catch errors. Both tools offer extensive libraries and documentation for learning. It's important to consider that Python's GIL model may be obstructive in handling intensive processing due to its locking mechanism. As such, Node.js may be more suitable for larger scale applications. Based on these observations, *asyncio* is best suited for I/O-bound applications like web servers using frameworks like *aiohttp*, network clients/servers, and applications requiring integration with synchronous Python code, while Node.js is ideal for I/O-bound and real-time applications like web servers, API servers, and applications involving a lot of network interactions. However, once again, personal preferences for Python or Javascript may impact the decision between which of the two to use.

6. Conclusion

For writing the application server head, Python's *asyncio* is a suitable choice. The *asyncio* library allows for TCP servers to be created for consistent and continuous processing of messages between clients and servers. Servers are able to efficiently connect to each other due to the asynchronous propagation of messages. Node.js, which is very similar to *asyncio*, is held back by Javascript's difficult error handling. While Java implements multithreading well, the language's strict compiler can lead to a frustrating development process. Additionally, the specifications for this project do not require Java's threading capabilities. Thus, *asyncio* is much more efficient for building flexibly single-threaded I/O-bound applications, like this server head. It's important to note, however, that *asyncio* may not hold up performance-wise for more intensive CPU-bound applications, due to multithreading limitations. In terms of choosing an approach for building this particular application server head, though, Python and its extensive library is an effective and reliable choice.

References

[1] *asyncio - Asynchronous I/O*. https://docs.python.org/3.9/library/asyncio.html.

[2] *Async IO in Python: A Complete Walkthrough*. https://realpython.com/async-io-python/.

[3] *Asyncio Vs Threading in Python*. https://www.geeksforgeeks.org/asyncio-vs-threading-in-python/.

[4] *Python vs Java: The Most Important Differences*. https://www.javacodegeeks.com/python-vs-java.html.

[4] *Difference between Python and Java*. https://www.geeksforgeeks.org/difference-between-python-and-java/.

[5] *Unraveling Python's Reference Counting: A Guide to Memory Management*. https://www.coderlang.com/python-reference-counting-explained.

[6] *What is Java Garbage Collection? How it Works, Best Practices, and More*. https://stackify.com/what-is-java-garbage-collection/.

[7] *Python Multithreading vs. Java Multithreading - Important Considerations for High Perfomance Programming*. https://topdeveloperacademy.com/articles/python-multithreading-vs-java-multithreading-important-considerations-for-high-performance-programming.

[8] *Intro to Async Concurrency in Python vs. Node.js*. https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36.