CS 131 Project: Evaluating Python's asyncio Module for Implementing Application Server Herd

Jonathan Chu
University of California, Los Angeles

Abstract

In today's age, there exist numerous frameworks and architectures for implementing web servers and thus several different choices of framework for this Wikimedia-style service we must design. Given that there will be frequent updates to articles, that various internet protocols will be required, and that clients will be largely mobile, a number of languages and frameworks could be suitable. In particular, this paper will focus on Python's asyncio library as an implementation choice and give some comparisons to node.js, another popular asynchronous run-time environment.

A prototype model was built using asyncio for an herd of five servers that support connections from clients and basic commands from them. Using the results from the prototype and other general features and characteristics of the asyncio library, this paper will outline some of the projected pros and cons for asyncio as a preference in the development of our new Wikimedia-style service. In summary, asyncio does provide the necessary features and protocols for our application and is a suitable choice for its implementation.

1. Introduction

With numerous options for server implementation, we begin the search by analyzing the application's specification. The Wikimedia-style service will be designed for news, where the servers will have to handle article updates more often, access to protocols outside of just HTTP is required, and clients will be largely mobile. As such, the Wikimedia Architecture may face performance and issues in receiving frequent updates, as it is not designed to handle such updates concurrently, and potential difficulties connecting via various protocols outside of HTTP.

The asyncio module in Python provides an infrastructure for single-threaded concurrent execution, access to sockets and other resources, and client and server models[1]. As such, asyncio seems to be a potential fit for our news application; it can process updates concurrently, access protocols outside of HTTP, and provide convenient implementation for our client-server model. The main idea behind asyncio is that since it is asynchronous and can handle multiple requests concurrently despite running on a single thread, we expect it to be more efficient at handling frequent updates to news articles than a synchronous runtime environment.

To assess the suitability of asyncio as the framework for the news application, this paper will evaluate it on a number of parameters, including performance, code readability, and documentation/community support. An application server herd prototype was built in order to better assess asyncio, with five named servers that communicate with each other and serve clients through TCP connections.

2. Prototype

The prototype, built with asyncio, is an application that receives, stores, and sends client location and time data. It supports five servers, named Goloman, Hands, Holiday, Welsh, and Wilkes, with hardcoded connections between these servers to indicate which can communicate with which. Each server handles commands from TCP clients, which send a number of commands which will be described in detail in the following sections, and the servers must respond to the clients accordingly. New data from clients must also be propagated to other servers via TCP connections. With so many connections and requests being sent and received among the servers, an asynchronous framework would intuitively be able to handle the load most effectively; we will explore this idea in the following sections.

2.1. Server Design

Upon entering a main event loop, each server actively accepts clients and reads input from them. Once received, input is processed and it's decided whether that input was a valid command. If so, the coroutine that handles the specific command is called and writes updates to stored clients and writes back a response, if necessary.

The implementation is modularized to each separate command and smaller tasks (fetch, parse latitude/longitude string, write response to client, etc.).

2.2. IAMAT Command & AT Command

Clients can send IAMAT commands to servers in the following format:

IAMAT <client_name> <latlon> <time_sent>

where client_name is the ID of the client sending the request, latlon is a string with the latitude and longitude coordinates of the client concatenated, and time_sent is the time, in seconds and nanoseconds, since 1970-01-01 00:00:00 UTC that the request was sent. Upon receiving an IAMAT command, the server first determines whether this command presents new data, either because it has not received data about that particular client before or because the data it has received on that client is less recent. If the IAMAT does not present any new information, the server does not process it. On the other hand, if the IAMAT does include new information, the server updates its own client data for that particular client and then responds to the client that sent the command using an AT command of the following form:

AT <server_name> <time_diff> <client_name> <latlon> <time_sent>

where server_name is the name of the server that received the command, time_diff is the difference between the time the command was received and the time it was sent, and the other parameters are the same as above. The server also pushes those same updates to all of the servers with which it talks to using an AT command of a slightly different form:

AT <client_name> <server_name> <latitude> <longitude> <time_diff> <time_sent>

where all parameters are the same as stated above, but latitude and longitude are separated for ease of parsing. If a particular server not running, an error message is simply logged and execution continues. Each of the servers that received the data then assess whether that data is new, and if so, updates its data on that client and pushes the updates to all of the servers to which it is connected. This process repeats until all servers identify that the data it received is redundant. Effectively, this process implements a flooding algorithm to push updates to all servers.

2.3. WHATSAT Command

In addition to IAMAT commands, which clients use to tell servers when and where they are, clients can also queue servers for information on other clients, via WHATSAT commands of the following form:

WHATSAT <client_name> <radius> <num_results>

where client_name is the name of the client it is requesting information about, radius is the radius around the client from which places are returned, and num_results is the number of nearby places to return. To handle the WHATSAT request, the server queries Google Places via an HTTP request to its API endpoint. asyncio does not have built-in support for HTTP requests, but a number of Python modules do, including aiohttp, one that does so asynchronously and is the one that was used. The server calls the API supplying the coordinates of the client, the radius, and an API key. Google Places returns a JSON result of all places within the radius of the client. Finally, the server responds to the client with an AT command of yet another variation:

AT <server_name> <time_diff> <client_name> <latlon> <time_sent>
<JSON_returned>

where all the data is the same as the response to an IAMAT request except following the line is the JSON returned by the Google Places API, formatted for readability.

The data returned by the Places API is not stored anywhere or reused; instead, servers query the API again each time a WHATSAT command is received.

2.4. Invalid Commands

When any command is received that does not fit the aforementioned prototypes, the server responds to the client in a message of the form:

? <invalid_command>

although in some cases where one of the parameters is not as expected, the method returns without responding with this invalid command or perhaps does not catch the error at all; this is one aspect of the implementation that could have been done more robustly, but this was a prototype after all and should not have to handle every edge case. For example, I did not check that a latlon parameter is within realistic bounds for latitude and longitude values or worry much about the case of requests being sent

from the far past. Nonetheless, I was able to build a prototype that features much of the functionality of the Wikimedia-style news application that needed to be tested with asyncio.

3. Evaluating Asyncio

Because no performance measurements were made of the asyncio runtime and there was no prototype of another framework that it could be compared to, it is difficult to assess the performance merits of asyncio as a runtime environment; we can only speak to its usability and readability with respect to the tasks that our news application will need.

3.1. Performance

As the application will need frequent updates to articles, the prototype was built to handle frequent location updates from clients via IAMAT commands and AT commands between servers. As mentioned before, there was not much performance testing done, but being an asynchronous runtime environment, we can trust that asyncio does a fair job at handling these many requests, at least compared to its synchronous counterparts.

3.2. Features

Depending on the API's and routines provided by any particular framework, it becomes clear what kinds of tasks it is better suited for. asyncio provides a number of asynchronous features, including an event loop, coroutines, and tasks[2]. It effectively abstracts details of the implementations of these features and simplifies the set of routines needed to be called by the programmer to just a few methods that provide core functionalities of the asynchronous environment. Even with these methods, it was somewhat difficult to develop the prototype because of my own unfamiliarity with the module and my reliance on documentation and examples. Objectively speaking, asyncio's methods provided a fairly simple way to define all tasks and functions needed while preserving concurrency, even if Python is not traditionally run asynchronously or even concurrently on multiple threads.

It is, however, worthy to note that the asynchronous nature of asyncio made it more difficult to program this server herd as opposed to synchronous execution and traditional routines that most developers are used to. The asynchronous execution and unfamiliar methods made it more difficult to understand the step-by-step execution of each server and thus made it more difficult to design and debug.

In addition to the above features specific to the asynchronous nature of asyncio, it also provides convenient connections for implementations of TCP servers and clients, which specifically serves the purposes of our application and is another feature that adds to its overall suitability.

Although asyncio does not provide an HTTP request feature, Python's rich collection of modules provides all the functionality our application could possibly need in sending requests and connections via network protocols.

3.3. Usability

Because Python is an extremely popular language familiar to a wide array of programmers, its syntax is fairly well-known and easy to pick up, allowing an easier transition for developers who don't have experience with asyncio already. Assuming Python 3 is installed, it is also very easy to set up asyncio, with a quick module installation via pip and an import statement. Also, Python's popularity is reflected in the widespread community support and documentation available on features of both the language and asyncio module, allowing for support throughout the development process.

Additionally, Python supports object-oriented programming that would make the application more modular and code more readable and error-free. Although I chose not to utilize classes in my prototype implementation, a larger application with more functionalities would benefit from such an approach.

The built-in data structures inherent to Python provided very straightforward and efficient ways to store and retrieve client data. Its dictionary easily mapped client names to all the data associated with them in a clear and efficient manner.

4. Comparison to Java

A number of features and specializations exist in Java that are not traditionally provided by Python, some of which could seem detrimental to asyncio as a choice for this application.

4.1. Type Checking

As Python is an interpreted language, it exhibits dynamic typing, where variable types are assigned based on runtime values and are able to change as those values change through time[3]. On the other hand, Java is compiled and statically typed, where each instance of each

variable has a type determined at compile-time[4]. The advantage to static typing that Java holds over Python is that more bugs will be caught at compile-time, leading to less type errors during execution which could lead to unexpected behavior.

In building and testing the prototype with Python, I did receive a number of type errors, since commands would come in as strings, but often I would need to process them as floats, and it was at times not straightforward to tell whether the particular variable was a string value or a float value at the time. This led to perhaps more type casts to floats than necessary, but the type errors that did come up were reported by the Python interpreter, and I was able to fix them.

Type-checking is definitely one concern that shows up when using Python, and it could be more or less serious depending on what data types show up in the news application that needs to be developed. However, in the prototype that required just strings and floats, the lack of type checking simply resulted in a few runtime type errors that were easily fixable.

4.2. Memory Management

Python and Java both have garbage collectors that operate automatically, meaning that objects don't need to be manually allocated and tracked like they would be in C/C++, but they do operate differently. In Python, all variables are allocated memory on the heap, whereas in Java, only objects are. Garbage collection in Python occurs when the reference count to an object becomes zero, meaning that no more variables point to the location and thus it is no longer needed[5]. Keeping track of reference counts does require extra space and operations, a slight downside to performance. However, Java has a generational method of garbage collection where objects are removed upon a mark-and-sweep[6]. In either language, memory management is abstracted from the developer and while Java may have a slight performance advantage over Python, the memory management models do not have an impact on implementation.

4.3. Multithreading

Java is known for its multithreading capability, as governed by the Java Memory Model. Multithreading in Java provides improved performance through concurrency. Python, on the other hand, can support multithreading, but its efficiency is limited by the Global Interpreter Lock[7], and Python is not generally a prime candidate for multithreading implementation. That being said, asyncio is likely a better candidate for our application with respect to performance than a multithreaded implementation in Java because we are implementing web servers that receive lots of requests. Multithreaded programs are great for concurrently executing tasks that are to be run on the CPU, but asynchronous environments are better suited for web servers that have to handle lots of network I/O requests, since a multithreaded program would not be able to receive these requests as efficiently as an asynchronous one would, even if the actual handling of the request once received could be faster in a multithreaded environment. For this reason, it isn't a concern that Python is not suited for multithreaded applications.

5. Comparison to node.js

node.js is the popular event-driven, non-blocking I/O runtime environment for JavaScript[8]. It provides many of the same functionalities for JavaScript that asyncio does for Python, primarily including built-in asynchronous processing capabilities. A parallel can definitely be drawn between the asyncio module and node.js environment, as they both serve to provide concurrent, asynchronous models for two of the most popular programming languages; also, both are often used to implement web servers that handle many requests.

As both node.js and asyncio are popular asynchronous environments for popular languages, there is a great deal of community support and documentation for both. In addition, both Python and JavaScript have countless well-maintained libraries to aid in the implementation of any application. Python's pip and node.js' npm both allow for single-command installations of modules, and Python's import statement and node.js' require statement load the modules in one-liners.

One difference between the two is that node.js frameworks are inherently asynchronous, and any methods defined will execute asynchronously unless otherwise specified. However, in asyncio, methods are synchronous unless specified as coroutines (with "async def"), so there will be implementation differences in that asyncio will require the developer to explicitly specify where the event loop occurs and what coroutines are awaited.

Without explicit testing, it is difficult to assess whether node.js would be a more suitable choice than asyncio or vice versa. Both are built for practically the same purpose and provide many of the same functionalities. The difference might come in performance, but without a side-by-side comparison of programs, it is hard to say which would be more efficient.

6. Conclusion

The application server herd, needing to receive and propagate lots of requests, should almost certainly be implemented using an asynchronous framework, as event-based concurrency handles large volumes of incoming web requests as well as today's web frameworks can. With this in mind, Python's asyncio module is, in my opinion, among the top choices for implementing such an application, since it does provide asynchronous event handling in one of the most popular, best supported programming languages of today. Whether asyncio is the single best choice of framework is a question that would require more thorough and varied testing, as there are suitable alternatives like node.js.

7. References

[1] "18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks¶." *4. More Control Flow Tools - Python 3.6.5 Documentation*, docs.python.org/3/library/asyncio.html.

[2] "18.5.3. Tasks and Coroutines¶." *4. More Control Flow Tools - Python 3.6.5 Documentation*, docs.python.org/3/library/asyncio-task.html.

[3] "Dynamic Typing." *MDN Web Docs*, developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing.

[4] "Static Typing." *MDN Web Docs*, developer.mozilla.org/en-US/docs/Glossary/Static_typing.

[5] "Memory Management Reference." *1. Overview - Memory Management Reference 4.0 Documentation*, www.memorymanagement.org/mmref/lang.html.

[6] Kumar, Pankaj. "Java (JVM) Memory Model - Memory Management in Java." *JournalDev*, 2 Apr. 2018, www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java.

[7] "Global Interpreter Lock." *ForLoop - Python Wiki*, wiki.python.org/moin/GlobalInterpreterLock.

[8] Foundation, Node.js. *Node.js*, nodejs.org/en/.