Exploring Python’s Asyncio Module For Server Herd Implementations

Arnav Garg
304911796
University of California, Los Angeles

Summary

Python’s *asyncio* is a library that allows developers to write single-threaded concurrent code using the async/await syntax and offers high-performance network and web-servers, database connection libraries, distributed task queues, etc. [1]. *asyncio* allows for asynchronous function calls, which means it gives developers the ability to handle multiple concurrent requests using a single thread by using the idea of preemptive scheduling.Therefore, most of the calls made through the module are non-blocking, which allows for the concurrent access.

The module provides a great platform for IO-bound and high-level structured network code that is ideal for server herd implementations. It allows servers to easily communicate with each other asynchronously (for example, via controlled flooding algorithms and its variants), reliably, and with high performance at scale. *asyncio* also offers flexibility in terms of network protocols beyond HTTP, including TCP/UCP and abstract base classes that can be used to implement network protocols [2].

1. Introduction

Over the last two decades, there has been a significant increase in the number of web server frameworks and architectures due to onset of more complex requirements in terms of reliability and speed. In addition to these, the need for scale and high performance have also contributed to the creation of concurrency, multiprocessing, multithreading and asynchronous I/O as ways to meet these demands.

With the increase in the variety of web server implementations, it becomes important to explore these different options to see what would work best for a given use case. In this paper, the focus will be placed on a Wikimedia-style service that is designed for news, where (1) updates to articles will happen with high frequency, (2) access will be required via various protocols, not just HTTP, and (3) clients will tend to be more mobile [3].

This paper will focus on exploring Python’s *asyncio* module as a suitable choice for server herd implementations, and in particular, of the Wikimedia-style service described above. In particular, the paper assess the *asyncio* framework on the bases of code readability, performance, documentation, and the community support.

This paper will also briefly compare Python and the *asyncio* module to *Node.js*, an asynchronous event driven JavaScript runtime that runs on the V8 Javascript engine and is used extensively in production servers [4].

2. Server Herd Prototype | Wikimedia - Style Prototype

The server herd prototype was built in Python using *asyncio* and *aiohttp* that consists of a herd of five different servers that accept connection and simple requests from clients (simulating a mobile as needed for the Wikimedia-style service). The five named servers, Golomon, Hands, Wilkes, Welsh and Holiday, communicate with a subset of each other and with the clients using TCP connections.

The prototype itself is built to receive, store and transmit client location along with communication timings on the client and server side. The port numbers for each server along with which other servers they can talk to are hardcoded into the program. When a server receives a particular command (explained in the sections that follow) from the client via a TCP connection , it should serve the client with an appropriate response if it recognises the command and close the connection with the client. When a server communicates with another server via a TCP connection, it should pass on all of its client information to that server, and the recipient server should update any previously recognised clients if a new communication was made to the donor server. This process must continue throughout even one of servers in the network fails.

Given the nature of the task we’re trying to achieve with many open connections across clients and servers, it makes sense to use an asynchronous framework since it will probably handle the load most effectively without blocking parallel/concurrent needs for communication.

2.1 Server Architecture

When a connection to server is established successfully, it begins to log messages in a dedicated server log with the name ‘*serverName.log*’ and starts an event loop that loops infinitely. Given the nature of the asynchronous I/O, the asyncio.start_server() method used to start the server/establish a connection with the server requires a callback function that is triggered once the server is started.

The callback function is used to to read and write to and from the server for the rest of the time while the server is running. When the server receives input from the client, it reads it and decides if the command is valid. If it is valid, the callback function triggers various coroutines that validates, processes, and generates an output for the input command. Depending on the command, it may also save details about the client and pass it on to other open servers at a future instance in time. It also handles incorrect commands with an appropriate response. The main logic for processing commands is modularized into a large set of coroutines, each of which is responsible for processing, validating or querying the input, or generating an output.

2.2 Commands

The client is allowed to send two types of commands - IAMAT and WHATSAT, while the server responds with the AT command in both cases. A server is allowed to send an ECHO command to other servers, which may in turn respond with ECHO commands of their own.

2.2.1 IAMAT, AT and ECHO commands

Clients can send an IAMAT command to a server using the following format:

IAMAT <clientName> <Location> <sentTime>

Here, the clientName refers to the client ID of the client sending the request, Location refers to the GPS location of the client at the instance it sent the IAMAT command request (represented as a combined string containing both the latitude and longitude), and sentTime refers to the time the message was sent by the client, which is represented in seconds and nanoseconds as the time since 1970-01-01 00:00:00 UTC.
When the IAMAT message is received by the server, the server validates the IAMAT message to ensure it is of the right format, with special emphasis placed on the number of tokens in the command, and on the format of the coordinates. If the server does not recognise the client, it updates the client's information which constitutes the client's name, location and sent time. It then formulates a response using the AT command using the format:

AT <serverName> <timeDiff> <clientName> <Location> <sentTime>

Here, the serverName is the name of the server that the client is currently connected to, timeDiff is the difference in time between when the client sent the message and when the server received the message, and in some ways represents the latency of the transaction between the client and the server. timeDiff can be positive, representing a lag between time sent by the client and time the message was received by the server, or negative, representing a time skew between the server and client's time, where the client's time is ahead of the server's time. The clientName, Location and sentTime are repeated from the original IAMAT command request made by the client in that order.

An IAMAT command also triggers a coroutine called *floodServers*, which tries to pass on this client's data to all other server's in the network that are currently running with which it communicates. The messages sent in the *floodServers* coroutine are of the format:

ECHO <clientName> <Location> <sentTime> <receivedTime> <sendingServer'sName>

Here, the clientName, Location and sentTime stay the same as those passed in the IAMAT command. receivedTime represents the time that the current server received the message, and sendingServer'sName represents the name of the server sending out the ECHO command to other servers. If a particular server with which the current server is trying to communicate and relay this command is not currently running, it logs this message into the log file and it moves on to the next server. On receiving an ECHO command, a server checks whether it knows the client, and if it doesn't, it saves it to its list of known clients and continues to the flood this message through another ECHO command to other servers it is allowed to communicate with. If it already recognises the client, it compares the last known sent time of this client's message with the new sent time passed in, and if the new sent time is more recent, then it updates the client and floods other servers, otherwise it ignores the ECHO command.

2.2.2 WHATSAT command

Clients can also query the server for information about other clients that are known to the server using the WHATSAT command in the following format:

WHATSAT <clientName> <radius> <limit>

Here, clientName represents the name of the client the querying client would like to know more about, radius represents the the radius around the queried client's last known location from which places will be returned, and limit represents the number of results to be returned. To determine the places in the given radius around the queried client's location, the WHATSAT command triggers a coroutine that queries the Google Places API via an asynchronous HTTP request to its API endpoint. This is done using *aiohttp*, another python module that can be used along with *asyncio.* The GET request passes in the API key, the client's location and the desired radius to the endpoint, which returns a JSON dump of the places it can find using these parameters. The server then responds to the client using an AT command with the following format:

AT <serverName> <timeDiff> <clientName> <Location> <sentTime>
<JSON_dump>

The parameters for the AT query are the same as those returned in the response for an IAMAT command. The JSON_dump contains the information returned by the get request, formatted and prettified for readability and limited to 'limit' (from the original WHATSAT request) results.

The server does not store the places returned by the GET request. Instead, the client must make another query each time the WHATSAT command is detected from a client.

2.2.3 Invalid Commands

In the instance that none of the above listed commands are detected or are detected in the incorrect format, the server always responds with:

? <invalid_command>

This covers almost all edge cases in terms of command combinations and parameter validation for each of the commands above. However, one edge case that was not considered was the actual bounds for the values of latitude and longitude, because this is an edge case and the goal is to build a prototype to explore the functionality and feasibility of python and the *asycnio* framework for a Wikimedia-styled service.

3. Assessing Python's Asyncio Module

It is difficult to assess asyncio in terms of runtime performance because we made no measurements as such, except for the latency between the time a message is sent and the time a message is received. However, it is difficult to make comparisons because of the nature of asynchronous I/O, where calls can be made in a different order each time for the same set of inputs. However, on average, it appears that asynchronous I/O is incredibly fast, responding the client messages before the blink of an eye.

3.1 Popularity

Before beginning looking into the deeper aspect of Python's asyncio module, it is important to consider Python's widespread popularity as a programming language over the last half decade as a reason for the success of *asyncio* as a module and a reason to adopt it as the framework for the Wikimedia-style service. Python has a easy learning curve, is very intuitive for programmers who are just starting off and very easy to adapt to syntactically for experienced programmers. Although not pre-installed with the Conda package management system or MacOS's homebrew, it is very easy to install the package and just needs a single command on the terminal using PIP (Pip Installs Packages).

Python also has a really extensive community on Stack-Overflow and extensive documentation online. It also offers the ability to use object-oriented programming which can be very helpful in terms of building modularity throughout the code. Additionally, and probably one of the biggest reasons for Python's popularity, is that it has automatic garbage disposal along with a very wide variety of data types.

All of these factors not only contribute to Python's success, but also contribute to the success of asyncio as a python module because it helps increase the usability and flexibility of what asyncio can do as a Python module.

3.2 Performance

In the Wikimedia-style service, it is important that frequent updates can be made to each of the articles. In the server heard prototype built and described above, the prototype appears to handle frequent location updates from clients via IAMAT commands and floods them to other servers via ECHO commands. While it is difficult to assess the performance of each of these calls, we know it is faster than a synchronous system since synchronous calls are blocking, preventing other subsequent function calls from being made and therefore taking a longer overall time for completion.

3.3 Features From Documentation

A framework's usage and appropriateness for a given task can be best judged by the APIs and libraries it provides. *Asyncio* provides an extensive set of asynchronous functions and features, mostly comprising of Tasks, which form the heart of asynchronous functions, along with event-loops, coroutines, callback functions and client-server models. Unlike many other languages such as C/C++ where establishing TCP/UDP connections requires the use of highly convoluted APIs that require manual socket creation, binding and listening, *asyncio* abstracts the lower level functionality and API methods from us, essentially serving as a higher level SDK with simple functions. While server based computing is not a trivial task for someone who has never done it before, the steps required to get a server running as well as the extensive documentation provided for *asyncio* help tremendously. It is also interesting to note that while Python programs are not normally multithreaded, it allows to write concurrent code and provides an easy way to define and use Tasks and async functions.

3.4 Usability

Compared to the predictable nature of synchronous function calls and normal imperative programming, the idea of asynchronous functions can be a bit harder to work and requires a small learning curve. This is because it is difficult to understand the nature of asynchronous calls, because they are non-blocking and pass up their resources to the compiler during off-time when they're waiting to return from another function. However, while there is a small learning curve, the power that is obtained from scheduling asynchronous calls in terms of freeing up the compiler for other necessary blocking calls is tremendous and outweighs the small learning curve.

4. Comparison to Java

4.1 Memory Management

Python and Java both have garbage collection via garbage collectors, so memory does not have to be manually allocated, reallocated or freed like C/C++. In either case, memory management is abstracted away from the user. In Python, all variables are allocated memory on the heap instead of the stack, while in Java, only Java objects are allocated to the heap. Python is particularly interesting in that its garbage collector is triggered when the reference count/link count of an object becomes zero, that is, no more variables point to the location/address of the variable in consideration, and therefore it is likely not being used and is garbage. However, it requires extra effort to keep track of link/reference counts for each of the variables, and therefore has a slight delay in computations and overall performance.

Java uses the mark-and-sweep algorithm for garbage collection, which is the traditional way most programming languages deal with garbage. This is slightly slower because it requires a two-fold process where the first phase involves marking all objects being used by using DFS, and the second phase involves sweeping all unmarked objects.

4.2 Type Checking

Python exhibits dynamic typing, which means that the type of variable is determined at runtime rather than during compile time. This means that a type mismatch will only be detected during runtime. However, Java uses static typing, which means that variable types are checked during runtime, including all methods that are being called on the variables to ensure that they are allowed on a variable of that particular datatype. Therefore, not only are errors known in advance during compile time - it also guarantees that fewer runtime errors will be caused because of the lack of type mismatches, and this provides more guarantees in terms of behaviour.

One of the errors I ran into during compile time is that I kept forgetting that location was a string, so I would try to use numeric methods on it and could only catch them during runtime. I also often ran into errors when generating the server return string, because the time was of a time datatype and not a string. However, the compiler/interpreter is really good at reporting errors and it was easy to debug.

4.3 Multithreading

Traditionally, many big technology companies have used Java on their backends. A big reason for this is that Java is known for its multithreading capability, courtesy of the Java Memory Model (JMM). The ability for easy multithreading provides inbuilt support for concurrency and parallelism. Python can support multithreading, especially through libraries like Pool. However, its efficiency is severely limited by the Global Interpreter Lock[5]. Therefore, Python is generally not used for multithreaded programs.

While this may indicate that Java might suit our needs better for the Wikimedia-style service, asyncio is better with respect to performance because of the nature of our program where it receives a lot of requests. As seen in traditional computing, multithreading is usually efficient for tasks on the CPU/GPU, but not as good at dealing with multiple concurrent requests in real time which is what occurs on web servers. This is largely because multithreaded programs are not efficient in receiving and handling multiple concurrent requests. Additionally, multithreading can cause races, which may not work in our case if the client messages more than one server concurrently with different sent times. To deal with this, we would have to set up critical sections using locks, but this would require more careful server development on the developer end and is more prone to design errors.

5. Comparison to Node.js

As described on the Node.js webpage, it is an asynchronous event driven JavaScript runtime that runs on the V8 Javascript engine and is used extensively in production servers [4]. Node.js extends javascripts functionality, by extending its use from a front-end client-side language to one that can now only be used on the backend for server creation. This is powerful because Javascript is one of the most popular and well known front-end languages used for web development, which automatically uses increases the support available from the javascript community and the extensive documentation available.

In terms of keywords used in its description, it is very similar to asyncio - it provides asynchronous methods and functions, and is event driven so possess the ability to use callback functions for I/O. The use of the V8 engine is important because most of the javascript that webpages generate or use for their front-end also uses the same V8 engine, which makes it easy for clients and servers to be setup and communicate with each other using the same language and do so very quickly. Therefore, asyncio and node.js are very comparable in terms of what they provide a developer. Javascript and Python both use dynamic type checking, and are therefore also very comparable in that aspect.

A quick look through node.js documentation [6] highlights an extensive range of easy to download and use node modules using the npm package installer, which is very comparable to Python's PIP package installer.

A key difference between node.js and python is that node.js is inherently asynchronous - this means that all function calls are automatically made to be asynchronous, while we would need to use Python's asyncio library to make and serve asynchronous requests. This makes it harder to program because the developer has to remember to use async and await function calls, and constantly differentiate between when an async method is required versus when a normal function call is required. A python developer will also have to explicitly create and start event loops, as well as ensure that all methods run to completion for execution finishes, which is automatically managed in node.js.

It is difficult to assess whether node.js will be better than asyncio in terms of runtime performance unless we setup a similar client-server herd architecture using javascript for the client and node.js for the backend server. However, a quick overlook of node.js suggests that this would be an interesting architecture to look into for our given use case since it appears to provide similar functionality with a more enjoyable development experience.

6. Conclusions And Further Recommendations

Overall, based on the exploration of Python's asyncio module, Java and Node.js, it is evident that the Wikimedia-style service should use an asynchronous framework for its server herd implementation. This is because of the large number of requests that can be made concurrently. This is also because asynchronous frameworks provide event-based concurrency, which handles a large number of requests efficiently and almost seemingly concurrently. Keeping this in mind, I would personally recommend an asynchronous framework that uses asyncio as one of my top choices to build the server herd to be used for the Wikimedia service. This is because of its incredible support, easy learning curve, extensive libraries, wide range of datatypes, automatic garbage collection, faster compile and server deployment times due to dynamic typing, and because of its asyncio module that provides a wide range of asynchronous I/O methods that are easily accessible to use and get accustomed to. I would also recommend trying to build a node.js implementation of the same model and running runtime performance tests, as well as extensive profiling in terms of memory before making a final decision on the choice of asynchronous framework to be used for this use case.

7. Citations

[1] "Asyncio - Asynchronous I/O¶." *Asyncio - Asynchronous I/O - Python 3.7.3 Documentation*, docs.python.org/3/library/asyncio.html.

[2] "Transports and Protocols¶." *Transports and Protocols - Python 3.7.3 Documentation*, docs.python.org/3/library/asyncio-protocol.html.

[3] "Project. Proxy Herd with Asyncio." *Project. Proxy Herd with Asyncio*, UCLA , web.cs.ucla.edu/classes/spring19/cs131/hw/pr.html.

[4] Foundation, Node.js. "About Node.js." *Node.js*, nodejs.org/en/about/.

[5] "Page." *GlobalInterpreterLock - Python Wiki*, wiki.python.org/moin/GlobalInterpreterLock.

[6] "Npm." *Npm*, www.npmjs.com/.