Using Asynchronous *asyncio* Framework for Server Proxy Herd

Lawrence Chen, *University of California, Los Angeles*

Abstract

When creating a server that will need to handle frequent connections effectively and flexibly from different mobile clients, a consideration that is important is how to reduce the bottleneck at the server level. One approach is to utilize asynchronous and concurrent programming. To explore this concept further, we have created a prototype for a server herd that utilizes Python's *asyncio* framework library to provide asynchronous networking capabilities. In this paper, we will look into the pros and cons of this library and the Python language in the context of this architecture, and compare them with the features of the Java language. We will be examining the two languages' characteristics in type checking, memory management, and multithreading. We will also briefly summarize the similarities between *asyncio* and *Node.js*, an asynchronous runtime built for Javascript. We ultimately conclude with a recommendation that Python and *asyncio* is a suitable framework for the application in question, because it is easy to use and provides a scalable solution comparable to Java and similar functionality to *Node.js*.

1. Introduction

Consider the Wikimedia architecture that utilizes a LAMP platform consisting of GNU/Linux, Apache, MySQL, and PHP using multiple, redundant web servers behind a load-balancing virtual router. However, if we wanted to extend this to build a new Wikimedia-style service designed for news, we would need to reconsider our architecture. For example, updates will happen more frequently, it should be able to handle various protocols, and we would need to be able to handle more mobile clients, such as those on cell phones. The old Wikimedia architecture with its central server will become a bottleneck, so we instead explore the idea of an application server herd. A server herd will allow parallelizable servers to handle higher amounts of request and add servers to be more conveniently closer to the proximity of mobile device clients.

To aid in our considerations, we prototype this architecture by building a proxy for the Google Places API[1] and utilize *asyncio* with Python 3.7.2. The service will take in IAMAT commands from mobile clients specifying a client's updated location via TCP connection with one of the servers in the server herd. This is expected to happen really frequently, as clients update their new GPS locations. Because we are utilizing the server herd, a client only needs to contact the single nearest application server with its IAMAT update. The server herd then propagates updated information of client locations by flooding all the servers, so all servers can stay updated on the rapidly-evolving data. At any point, a client can also query with WHATSAT for nearby places of any client's location. The contacted application server will respond using the client's most recent location update. Each application server is capable of contacting the Google Places API to get the "nearby places" information, thus the API is representing the database server that is used to retrieve the more stable data.

We observe that writing the application with Python and *asyncio* is quite easy. The server.py file is very lightweight, and the server herd is easily extendable by simply adding a new server in the bidirectional "talks-to" graph. Thus, with Python's easy to read syntax and *asyncio*'s simple interface, maintaining the application does not appear to be a problem. Although the type checking for Python is a reasonable concern, it is not a crucial problem that cannot be addressed. In addition, we will see that Python's memory management and multithreading approach actually suits the application very well for frequent updates from clients and servers.

2. Suitability of asyncio Framework

We will now examine the *asyncio* framework more closely in the context of implementing an application server herd to observe how it is suitable for our purposes.

2.1. What It Is and How It Works

Asynchronous I/O capabilities in Python 3 were first proposed in PEP 3156, calling for an *asyncio* module[2]. *aynscio* is an event-driven asynchronous framework that allows us to easily write concurrent code utilizing async/await syntax[3]. The built-in functionality of *asyncio* provides the ability to run Python coroutines concurrently. This is built upon *asyncio*'s event loop, to schedule and run tasks, futures, and callbacks, such as the aforementioned coroutines and I/O intensive procedures.

Coroutines can by declared with async/await syntax, which means it can be asynchronously scheduled in the event loop, and use `await` to voluntarily yield its execution as it waits for a callback from the finished function call. For example, this allows the event loop to concurrently schedule other coroutines when it is waiting on an I/O task to complete. We can use coroutines with *asyncio*'s event-loop simply by calling `asyncio.create_task()` to wrap a coroutine as an *asyncio* `Task` and allow the event-loop to schedule for its execution.

The coroutines and its ability to yield execution is useful for our server herd because of the expected frequent I/O that it would need to perform. The framework supports TCP, SSL, UDP network connections, which helps satisfy a requirement that our server needs, which is the ability to handle various network protocols. This is done by using *asyncio*'s `Stream` primitive, which is a high-level async/await-ready primitive that allow sending and receiving data without callbacks or low-level protocols and transports[5]. Establishing a TCP network socket is as simple as calling the `asyncio.open_connection()` function. (Python also provides other easy to use modules to help with HTTP, such as *aiohttp*, which we use to contact the Google Places API.)

2.2. Applying to Server Herd

With the basic functionalities of *asyncio* mentioned above, we can quickly apply asynchronous programming to our server herd prototype and see how it is really suitable for our purposes. Each application server in the herd needs to be able to handle frequent I/O connections from both clients and other servers, and output corresponding information where necessary. By utilizing coroutines to handle each of these connections on an event loop, we can process each request quickly and efficiently.

We simply have an event loop that runs our server by calling `asyncio.start_server()` which takes a callback function to handle all new requests. This function can be a coroutine which is automatically scheduled as a task whenever a connection is made to our server. Therefore, our server will have an event loop in charge handling the frequent location updates, propagating information throughout the herd, and requesting data from Google Places API effortlessly. As we can see, creating the server or creating concurrent tasks are very simple with *asyncio*, often requiring only a single function call.

2.3. Performance

For our use case, in which a majority of actions our servers need to handle is I/O, this concurrent approach performs incredibly well. By utilizing coroutines that allow `await` keywords, our tasks can voluntarily yield whenever an I/O intensive action is being done. The server requires very little CPU time, only to parse commands and store location information. The majority of the server's functionality, e.g. when receiving from or responding to clients and flooding location information to other servers in the herd, these are all I/O actions that our coroutine can yield on, allowing other coroutines to begin handling a new request. Therefore, our coroutines are rarely being blocked and no preemption is necessary. Compared to a true multithreaded or multiprocessing approach, our server herd is incredibly lightweight, with coroutines requiring much less resources and no need for expensive preemption, because tasks are frequently yielding its execution during I/O actions. We see that *asyncio* is really suitable for creating a new application server to be added into the server herd by simply inserting itself into the flooding graph, thus solving the requirement of handling mobile clients. In addition, by using the event loop, we can handle the frequent updates that we will expect.

3. Python vs. Java

From the previous section, it is clear that *asyncio*'s coroutines and event loop allow us to effortlessly create a new server in the server herd to handle the frequent requests that our server will receive. Now we will compare the Python language itself to the Java language, and see how the type checking, memory management, and multithreading characteristics of Python helps it remain comparable to Java.

3.1. Type Checking

A reasonable concern when moving the Python prototype and scaling it to be production level is the type checking of Python. Both Python and Java are considered strongly typed, which means explicit conversions are necessary when coercing values to different types. But, unlike Java which is statically typed, Python uses dynamic typing[6]. This means programming in Python will need to utilize strategies such as duck typing (to check for the existence of certain methods before calling them) or use try/exception clauses. These two approaches can help prevent the server from completely crashing during runtime, but it does not guarantee our code is bug-free when we start the server.

On the other hand, Java's static typing means we would know of any type checking errors during compile time before deployment, so we can avoid unexpected errors. This makes Java more reliable in this aspect. If this is a strong concern as the code base becomes too large to manually maintain, there are solutions to make Python's type checking more static. One solution is to use an open-source Python module called *mypy*[7], which can be integrated into unit tests and development workflow to help statically type check Python code. In addition, certain IDE's such as PyCharm or editors with downloadable plug-ins could also provide Python static checking features[7].

For the purposes of this server, because of how lightweight Python code is and the simple interface *asyncio* provides to easily create a server herd, type checking may not be dealbreaker for using Python as a whole since the code base would be relatively manageable. By using try/exception or duck typing along with extensive unit tests, we could allow our server to keep running without crashing and continue to handle the frequent requests despite possible bugs. Finally, third party and open-source solutions make it so that type checking is not that problematic. So while in this aspect Java is much more reliable, there are solutions that make Python manageable and reasonable despite its dynamic type checking nature.

3.2. Memory Management

The memory management of Python is mostly reliant on reference counting for its garbage collection[8], while Java utilizes mark and sweep algorithm[9]. Technically, it is possible for both languages to implement some version of the opposite garbage collection method. For example, Python could implement mark and sweep in its low level implementation or convert Python bytecodes to Java bytecodes and run the program in a JVM. In addition, Java has different variations of mark and sweep, and some versions of JVM incorporate reference counting as well. Therefore, if difference in the memory management is really significant, a solution could always be for either languages to implement the desired management approach, albeit not being convenient. For the sake of simplicity in comparing the two languages, we will look at the basic vanilla versions of Python and Java's memory management approaches. This means essentially comparing reference counting to mark and sweep.

Mark and sweep algorithm works by going through the heap starting from root blocks of memory and iteratively marking all blocks of memory that are referenced and can be accessed starting from the root. Then, the algorithm sweeps through the heap, freeing up all memory blocks that have not been marked and are thus no longer referenced to[10]. This sweep across the heap twice could be inefficient, especially if Java pauses normal program execution while garbage collecting, which is a concern for our servers that are expected to handle frequent updates rapidly[9]. However, variations to mark and sweep, like concurrent mark and sweep, generational mark and sweep, or even running the garbage collector on a separate thread could help alleviate this issue. Mark and sweep is considered a robust standard, since it is able to handle cyclic references and does not incur additional overhead when allocating or deallocating objects.

On the other hand, reference counts in Python works by incrementing a counter for each object every time a new reference to that counter is created. The counter is decremented whenever a reference is no longer reachable or is out of scope. Whenever the counter for an object reaches 0, the object is recycled and deallocated. An advantage of this is that the garbage collection is real time, and any objects that are no longer referenced will not linger in the heap taking up memory. The garbage collection is determined by its reference counter, so it is deterministic and predictable, unlike Java, in which when the garbage collection is triggered is not always clear. This is useful for our server herds, because we need to be able to handle large volumes of traffic easily. Deterministic memory management would be beneficial as we scale. Each of our application server will need to continually create and drop multiple connections with clients and other servers. Reference counting and its real time deterministic garbage collection is perfect for handling these connection and stream objects that have high turnover rates. Each object might have a short lifespan, and is immediately recycled once it is used up instead of taking up space in the heap. The disadvantage of reference counting is the small overhead of incrementing/decrementing object counters when using them, and also the inability to clean up cyclic references. However, with the well-defined functionalities of our server, we can simply avoid any cyclic references. Thus, for our purposes of high frequency objects, reference counting's advantage of deterministic real time collection outweighs avoidable cyclic reference concerns.

3.3. Multithreading

Java utilizes allows for true parallelism by using multithreading. This allows programs to split up parallelizable work into multiple threads to work at the same time, greatly improving performance and throughput. Using multithreading, we can maximize the throughput of a CPU or even take advantage of multiple cores. If we were to utilize Java's multithreading, each thread could handle a separate request and could ideally update different parts of an object, such as our dictionary storing client information.

Python also supports multithreading, such as with the `threading` module that makes such functionality easy[11]. However, due to Python's Global Interpreter Lock (GIL)[12], Python cannot execute more than one thread at once. The GIL was used to help prevent race conditions on object reference counting for memory management. Therefore, only one thread is allowed access to the interpreter at one time. For example, this means there cannot be multiple threads writing to the client dictionary at the same time, which does decrease efficiency of our code. Essentially, multithreading utilizes scheduling for threads to run concurrently, but never truly simultaneously. While for a CPU-bound program, this lack of true parallelism might be of concern, our application is more heavily IO-bound. By using the server herd architecture, we have already "parallelized" the problem to a certain extent, allowing multiple servers to handle requests at multiple locations. However, for our application, the main bottleneck is in the simultaneous I/O-bound tasks that we need to frequently run, such as whenever we accept a connection from clients, contact the Google API, or propagate information to the other servers in our flooding algorithm. The CPU-bound tasks that our server ever really does is updating a dictionary of clients. Therefore, despite the lack of true parallelism in Python, because our servers function mainly as network proxies that are heavily reliant on I/O-bound tasks, we still get parallelism where we need it the most to avoid serious bottleneck issues.

4. asyncio vs. Node.js

We are also curious as to how *asyncio* compares with *Node.js* in their overall approach. Similar to *asyncio*, *Node.js* is an asynchronous event driven framework to build

scalable network applications and handle connections concurrently[13]. Both utilize an event loop to run tasks concurrently, but *Node.js* is single-threaded with multi-threading in the background for asynchronous tasks (similar effects can be achieved in Python with the `Thread` module). The event loops for both are extremely comparable, with coroutines and await functionality essentially being the same as asynchronous callbacks. According to their website, *Node.js* treats HTTP as its first class citizen, but can also create TCP servers by utilizing their `Net` module. *Node.js* also utilizes promises and callbacks syntax, in addition to the async/await syntax, which is not really more than a syntactic preference or concern. As we can see, the use cases, functionality, and approaches are very similar for both Python's *asyncio* and Javascript's *Node.js*, especially for our IO intensive application. Ultimately, deciding between the two frameworks seem to come down to preference of the language itself, in which arguments could be made for both sides. *asyncio* is technically native to Python, while *Node.js* is a third party library. This means *asyncio* will probably have better documentation and support, and Python is known for its well-supported libraries. On the other hand, *Node.js* is a third-party Javascript runtime that simply allows Javascript to be run on servers instead of just on client side. However, some may find the idea of using the same language, Javascript, for both frontend and backend to be very appealing. So for our purposes, Python and *asyncio* are work just fine.

5. Difficulties and Problems

Prototyping the server herd was not met with too much difficulty. I did notice that the floating point precision in Python cannot handle more precision than what could possibly be reported for the time stamp. For example, `1520023934.918963997` would by truncated to `1520023934.918964`. This means some information may be lost, and if a client is able to update location within that time frame (although extremely unlikely), our server may not know which is the more recent update. In addition, if a server happened to be down when another server begins to flood location information throughout the herd, the server is not guaranteed to receive the new propagated information when it comes back up. However, this is technically okay as detailed in the specifications, indicating that there is no need to relay old messages to servers that come back online. Other than that, the prototyping went quite smoothly.

6. Conclusion

From our research, we conclude that Python and its *asyncio* library is a suitable framework for our Wikimedia-styled service designed for the news, specifically with high frequency updates, variable protocols, and mobile clients in mind. The asynchronous functionality provided by *asyncio* fits perfectly for the I/O actions that our server herd needs to perform, and the simplistic syntax makes adding new servers to the server herd very easy. *asyncio* supports different types of protocols such as TCP, UDP, and SSL, and Python also has modules that can support even more if necessary. While one may have some concern about Python's scalability for CPU-bound tasks, because of the nature of this application and its intensive use of I/O-bound tasks, Python's implementation of memory management and multithreading make it an optimal candidate for the application. Given the type of problem we need to solve, Python's performance is remarkably comparable or even equal to that of Java. With Python's simple syntax and easy to use *asyncio* library for creating servers for fast programming and iteration, along with its comparable performance due to the volume of I/O tasks, we strongly recommend the usage of Python and *asyncio* for this kind of application.

7. References

[1] *Google Maps Platform Documentation,* https://developers.google.com/maps/documentation

[2] *Asynchronous IO Support Rebooted: the "asyncio" module,* https://www.python.org/dev/peps/pep-3156

[3] *asyncio - Asynchronous I/O,* https://docs.python.org/3/library/asyncio.html.

[4] *Coroutines and Tasks*, https://docs.python.org/3/library/asyncio-task.html

[5] *Streams,* https://docs.python.org/3/library/asyncio-stream.html

[6] *The Ultimate Guide to Python Type Checking,* https://realpython.com/python-type-checking/#dynamic-typing

[7] *How to Use Static Type Checking in Python 3.6,* https://medium.com/@ageitgey/learn-how-to-use-static-type-checking-in-python-3-6-in-10-minutes-12c86d72677b

[8] *Memory Management in Python,* https://realpython.com/python-memory-management/#garbage-collection

[9] *JVM Garbage Collectors,* https://www.baeldung.com/jvm-garbage-collectors

[10] *Mark and Sweep: Garbage Collection Algorithm,* https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/

[11] *threading - Thread-based parallelism,* https://docs.python.org/3/library/threading.html

[12] *What is the Python Global Interpreter Lock (GIL),* https://realpython.com/python-gil/

[13] *About Node.js,* https://nodejs.org/en/about