Suitability of the `asyncio` Python Networking Library for Designing an Application Server Herd Architecture

Naim Ayat, *UCLA Department of Computer Science*

Abstract

The LAMP (Linux, Apache, MySQL, PHP) model of web service stacks allows for utilization of multiple, redundant web servers behind a load-balancing virtual router for reliability and performance. This archetypal bundle is frequently used in industry and serves as the basis for numerous globally-recognized sites, such as those that operate on the Wikimedia Architecture. However, LAMP quickly becomes a bottleneck when (1) sites are updated often, (2) access to servers is required via various protocols, (3) and clients are mobile. Therefore, designing an optimal Wikimedia-style service for news publication would require an alternate web stack, potentially one rooted in Python or Java. This paper explores the `asyncio` Python 3.6.4 networking library as a case study in the context of a prototype application server herd architecture – a parallelizable proxy for the Google Places API. A final recommendation is made in favor of a Python implementation over one in Java.

1. Introduction

The objective is to create a Wikimedia-style service for news; thus, it is necessary to determine the ideal framework for constructing a web server herd that will effectively withstand frequent and varied I/O interactions. Two primary approaches will be considered and evaluated. The first utilizes Python version 3.6.4 with the `asyncio` module. The second is based on Java 8.161 with the Node.js runtime.

In order to make an informed recommendation regarding the suitability of the `asyncio` framework for an application utilizing such an application server herd, it was necessary to develop a prototype herd for testing and observation. Hence, a simple a parallelizable proxy for the Google Places API was constructed with `asyncio` as its basis.

Using this development as a benchmark, further research was conducted on `asyncio` and Node.js, as well as the Python and Java languages themselves. The various approaches to constructing an application server herd architecture were compared in regard to several core concepts, including type checking, memory management, and multithreading.

1.1. Development & Testing Environment

Development was conducted remotely on the UCLA SEASNet Linux Server 09, an 8-core Intel Xeon CPU E5-2640 v2 at 2.00GHz with 64GB RAM. The Python version installed at the time of development was 3.6.4.

Preliminary testing was conducted locally on an Ubuntu virtual machine to prevent the domination of excessive resources on the shared SEASNet machine. Final tests, however, were transferred to and conducted on the aforementioned SEASNet Linux Server.

1.2. Prototype Overview

The study began with the creation of a simple application server herd architecture prototype. Using the Python language's `asyncio` module, client location data is communicated across five distinct servers with the following IDs: `Wilkes`, `Hands`, `Welsh`, `Goloman`, and `Holiday`.

The servers communicate bidirectionally with the following configuration:

- `Goloman` communicates with `Hands`, `Holiday`, and `Wilkes`.
- `Hands` communicates with `Wilkes`.
- `Holiday` communicates with `Welsh` and `Wilkes`.

Clients that emulate mobile devices send their location data to the server, prompting a server response including the difference between the server's idea of when it got the message from the client and the client's time stamp. It is also viable for clients to query for information concerning places "near" the locations of neighboring clients. "Near", in this context, is defined by a radius (in kilometers), which is passed in as an upper boundary parameter (simply denoted `radius`) to the Google Places API.

The prototype is implemented in a single source file, `source.py`, which is packaged within the directory containing this paper.

1.3. Prototype Usage

To initialize a server, enter the following command:

```
$ python3 server.py <serverName>
```

Here, `<serverName>` denotes one of the five accepted IDs outlined in Section 1.2.

2. Remarks on `asyncio`

2.1. Module Basics

The `asyncio` Python module provides an infrastructure for parallel processes in event-driven programs. It allows

for the development of single-threaded concurrent code while abstracting the majority of synchronization issues through the use of constructs such as event loops, coroutines, and futures [1].

In the context of this study – considering `asyncio` as a basis for a Wikimedia-style news service – servers must have low latency to accommodate for frequent and mobile client interaction. Further, low throughput is necessary to successfully serve multiple clients simultaneously. Therefore, the fact that `asyncio` allows for asynchronous I/O is promising. At face value, it seems that `asyncio` is inherently superior to a stack such as LAMP for the construction of an application server herd; it eliminates the constriction of only having the capacity to serve one client at a time.

2.2. Server Design & Abstraction

Though Python is not strictly either functional or object-oriented, `asyncio` is most efficiently implemented in an object-oriented environment. Hence, it is logical to create a `Server` class with the ability to accept connections and serve requests. In the prototype server, for example, `class Server` takes one argument, `asyncio.Protocol`, which is the base class for implementing streaming protocols. This allows the server to handle TCP connections. Protocols in `asyncio` drastically simplify the use of TCP, as they provide an interface for servers to implement. In other words, servers written in Python only need to function to handle requests; `asyncio.Protocol` abstracts lower-level TCP and IP intricacies. This is quite convenient for the programmer, as dealing with the fundamentals of TCP/IP quickly becomes tedious in languages like PHP – the basis of LAMP. The same drawbacks exist in Java. Therefore, Python is favorable in terms of ease-of-use for its heavy layers of abstraction.

There is, however, one issue with such an abstraction: a server cannot initiate a TCP connection on its own. Instead, it becomes necessary for the programmer to develop a dedicated initiation class that the server can utilize. It is worth noting, though, that this is a relatively small price to pay for the level of abstraction `asyncio` protocols offer.

The coroutine-based API that `asyncio` offers, `Streams`, consists of convenience wrappers that further simplify TCP interactions. For example, `open_connection` function wraps `create_connection` and returns a reader and writer tuple. Of course, this abstraction substantially facilitates the process of opening connections with socket programming in no-frills Python (or, for that matter, PHP and Java). Also, note that readers and writers can operate asynchronously, allowing for straightforward and efficient pseudo-concurrent programming.

2.3. Comparison of `asyncio` to Node.js

JavaScript offers the Node.js runtime, which, like `asyncio`, is an asynchronous event-driven non-blocking I/O model for server-side scripting. The two frameworks are quite similar in their implementation, though they do have some performance (and, of course, syntax) disparities.

Node.js implements the concept of a `Promise`, which allows for manipulation of values that are unknown at the time of `Promise` creation. Therefore, the programmer is able to associate handlers to an asynchronous process' return value [6]. This concept offers functionality similar to that of the callbacks offered by `asyncio`. In terms of benefit, a Node.js `Promise` is typically less verbose and not as burdensome to follow as an `asyncio` event loop.

In terms of performance, `asyncio` is guaranteed to faster than Node.js when tested on the same hardware. Note that Node.js and `asyncio` are both single-threaded and asynchronous; they do not benefit from the same performance amplification as multithreaded implementations on horizontally-scalable clouds with multiple cores. This concept will be explored further in Section 3.2.

For now, it is noted that Node.js' dependency on semaphores and its inability to access shared memory make it much slower than `asyncio` for applications with frequent I/O interactions. Hence, for the intention of creating a Wikimedia-style news outlet which will receive frequent updates from multiple clients, `asyncio` is preferable.

3. Remarks on Python

3.1. Memory Management

Perhaps the biggest draw of the Python language is its famed ease-of-use. This is due, in part, to the memory management model. The Python memory manager employs a raw memory allocator, which works in conjunction with the operating system memory manager, to create space in the private heap. In addition, object-specific allocators aid in initializing the same heap by considering the unique requirements of each type. Further, Python's garbage collector primarily operates on reference counting. This implies that there is low risk of garbage collection introducing significant overhead [4].

The Java Memory Model is similar, though its garbage collector introduces a slight variation that poses prob-

lems in the context of a server herd architecture. In specific, Java garbage collection is a synchronous operation that blocks program from running for a set time. These unavoidable pauses will nearly nullify the benefits of parallelism (or asynchronization) [3]. Hence, the Python Memory Model is superior for server construction.

Recall that the user has no control over memory in Python. This is advantageous in that the language becomes less verbose and easier to grasp; the glaring drawback is that the programmer cannot write situation-specific memory management code in order to truly optimize runtime and memory space. The lack of optimality is not a prominent issue in small programs such as the prototype `server.py`, but it becomes increasingly apparent as code scales. It is worth noting, however, that manual memory management also becomes a tedious issue as programs get larger; it is more likely that opportunities for memory leaks will be introduced. In this sense, the Python Memory Model is optimal for designing an application server herd.

3.2. Multithreading

Though Python supports multithreading, `asyncio` conveniently employs a single-threaded, asynchronous event loop. On a multiple-core machine, this is clearly suboptimal. In theory, the most efficient implementation of `server.py` would have been the one that employed distinct threads for different TCP operations such as read and write.

An advantage of the `asyncio` model, however, is that programs will perform the same across any system, regardless of a given machine's number of cores.

The Java language offers a much more sophisticated multithreading interface than Python, so the prior is best for situations wherein developing asynchronous (or plain single-threaded) applications is illogical – for example, if servers were to run on a large cloud of multiple-core machines. In a situation where the primary issue is horizontal scaling, it is clearly preferable to implement multithreaded servers rather than using `asyncio`.

However, multithreading introduces issues such as race conditions, deadlocks, and starvation. Asynchronous code eliminates the potential for these pitfalls altogether, so servers implemented in Python's `asyncio` are less susceptible to error.

3.3. Dynamic Type Checking

Python does not address types at compile time; rather, they are checked and addressed only during runtime. This introduces both complications and advantages for the programmer.

First, and most problematically – bugs will not become apparent until runtime. Though this is not a massive issue during development and testing, it is worrying for code that is pushed to production. For instance, it is possible that a bug is first caught by an end-user, causing unpredictable behavior or server crashes. The solution to this problem is rigorous quality assurance testing, though no QA team is infallible.

This drawback is heavily offset by the succinctness that dynamic type checking – and, by extension, the Python language – allows. Developers do not need to address concerns such as type declarations, typecasting, or argument types. Rather, code can be concise, readable, and uncomplicated.

Java, on the other hand, is statically typed. It reduces the probability of bugs occurring during runtime, though it does not eliminate it. Additionally, code becomes much more verbose due to the lack of any form of type inference. For example, what was achieved in under 200 lines in the prototype `server.py` would likely have taken just under double that in Java – making Python the preferred language when evaluated on the metric of conciseness.

4. Conclusion

With the aforementioned criteria considered, it is recommended that a Wikimedia-style news website be constructed with Python and the `asyncio` library as opposed to PHP (from LAMP) or Java.

Beyond the languages' disparities in memory management, multithreading, and type checking, Python is simply the most adaptable and unchallenging to implement. Of the most widely-used programming languages in modern computing [7], Python is likely the simplest to learn. Because of its tremendous readability, the language is ideal for group development – and hence a Wikimedia platform that will doubtlessly be edited and adapted by numerous different programmers via unique clients.

Regarding web development in specific, Python allows for painless parsing of JSON or XML-style data with its extensive set of string operations. Methods like `join`, `split`, and `replace` make the translation of web data to lists incredibly straightforward. The LAMP architecture boasts nothing of the sort, while Java only offers `split` (and, as of Java 8, a more convoluted and verbose version of `join`) [2].

Java, as previously mentioned, scales more effectively than Python for large projects. The latter language is also interpreted, making it slower to execute than many of its counterparts. On the other hand, Java tends to consume

more memory, fall victim to memory errors, and yield unpredictable behavior with the introduction of multiple threads.

Therefore, for the purpose of an application server herd architecture such as that in the prototype `server.py`, Python is the victor. For good reason, it is the preferred language of global web services such as Instagram, YouTube, and Spotify; as well as one of the official languages of Google. Its extensive libraries, cross-compatibility, and support for both imperative and functional programming make Python the most widely-used language in the Internet of Things [5].

Python is a language designed for usability and the power to adeptly handle high-traffic websites. With the help of extensive single-threaded network libraries such as `asyncio`, it has the upper hand over Java as the language of choice for the construction of application server herd architectures.

4. References

[1] “18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks.” *18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks - Python 3.6.4 Documentation*, docs.python.org/3/library/asyncio.html.

[2] “Java™ Platform Standard Ed. 8.” *StringJoiner (Java Platform SE 8)*, 19 Dec. 2017, docs.oracle.com/javase/8/docs/api/java/util/StringJoiner.html.

[3] Manson, Jeremy, and Brian Goetz. “JSR 133 (Java Memory Model) FAQ.” University of Minnesota Duluth, www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html.

[4] “Memory Management.” *Memory Management - Python 3.6.4 Documentation*, docs.python.org/3/c-api/memory.html.

[5] Miorandi, Daniele, et al. "Internet of things: Vision, applications and research challenges." *Ad hoc networks* 10.7 (2012): 1497-1516.

[6] “Node.js Documentation.” *Node.js Foundation*, nodejs.org/en/docs/.

[7] O'Boyle, Noel M., Chris Morley, and Geoffrey R. Hutchison. "Pybel: a Python wrapper for the OpenBabel cheminformatics toolkit." *Chemistry Central Journal* 2.1 (2008): 5.