CS 131 Project
Proxy Herd with *asyncio*

Puvali Chatterjee
UID: 504822474

Abstract

This report examines the suitability of Python's *asyncio* asynchronous networking library as a candidate for replacing part or all of the Wikimedia platform in a news service that is based on an application server herd architecture. Research for this project involved the implementation of a proxy server herd using *asyncio*. This report also compares Python and Java-based approaches to building this application, as well as a brief comparison of the overall approaches of *asyncio* and Node.js to tackling the problem of building this application.

I. Introduction

Wikipedia is an online encyclopedia that hosts 56 million articles - in all its language editions combined - and attracts more than 17 million edits as well as 1.7 billion unique visitors per month [1]. Wikipedia and other related websites are based on the Wikimedia server platform, which is based on Debian GNU/Linux, the Apache web server behind an NGINX-based TLS termination proxy, the Memcached distributed memory object cache, the MariaDB relational database, the Elasticsearch search engine, the Swift distributed object store, and core application code written in PHP+JavaScript, all using multiple, redundant web servers behind the Linux Virtual Server load-balancing software, with two levels of caching proxy servers (Varnish and Apache Traffic Server) for reliability and performance [2]. Figure 1 shows the flow of a webrequest through the Wikimedia server infrastructure.

Although the Wikimedia server platform is suitable for Wikipedia, it may not be the most efficient platform for a news application that would require more frequent updates to articles as well more mobile clients who would be accessing the service via several different protocols besides HTTP (or HTTPS). In such a scenario, Wikimedia's PHP + JavaScript core application server would create a bottleneck, and it would become immensely difficult to add new servers for mobile clients that would frequently broadcast their GPS locations.

Given the requirements of such an application, a different architecture consisting of servers that are able to

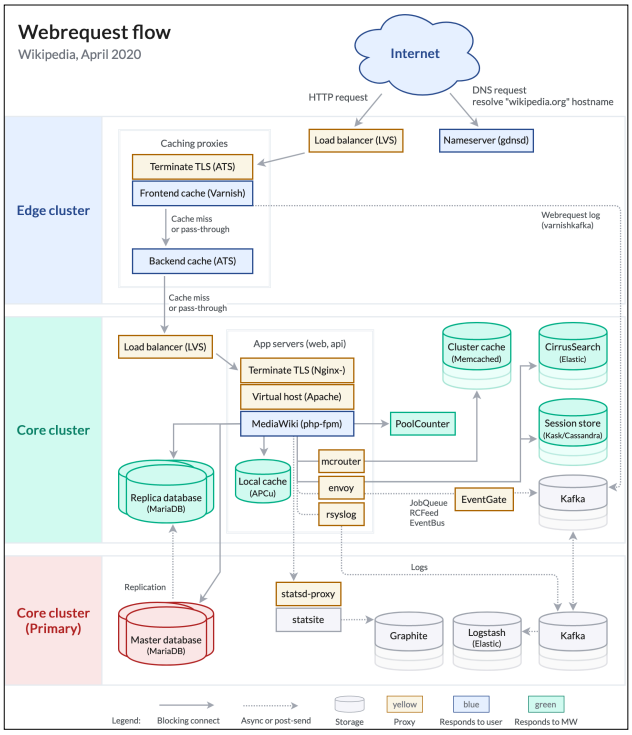

Figure 1: The life of a Wikipedia webrequest inside the Wikimedia server infrastructure, as of April 2020 [3]

communicate with each other *directly* may be a better choice since it would ease the bottlenecks on the core server. Such an architecture, termed an "application server herd" or simply a "proxy herd", would integrate interserver communications via TCP messages in a flooding protocol where application servers that can "talk" to each other will notify each other if a user's cell phone posts its GPS location to one of the servers. Thus, there is no need for the server to talk to the database - although of course all servers must be able to communicate with the core database and caches, and if a single server goes down - the other servers must still be able to function normally.

asyncio is Python's single-threaded asynchronous library for concurrent applications that incorporates cooperative multitasking, where the application itself can decide when to switch between processing indi-

vidual requests [4], so that tasks may explicitly hand over control or block. This event-driven nature of *asyncio* would allow updates from clients to be processed and forwarded to other servers in the herd rapidly.

II. Python and Java

In addition to rapid server updates, an *asyncio* approach to implementing a proxy herd must also be fairly easy to write. Furthermore, applications written with *asyncio* need to play well with other existing applications and be maintainable as well as reliable. With these goals in mind, following is a comparison of Python and Java-based approaches for this application.

2.1 Type Checking

The primary difference between type checking in Python and Java is that Python is dynamically typed whereas Java is statically typed.

Python container objects such as lists and dictionaries may contain objects of any type, and the same variable names can be assigned to objects of different types in the same program. Since the programmer does not need to know or declare variable types, Python code is more concise and easier to write, which is a great advantage while implementing a complex server herd application with multiple modules that share variables.

However, this may introduce ambiguity that could make the code less readable, which is important for maintainability of the program long-term. Furthermore, since types are only checked by the Python interpreter during runtime, any type errors that would cause the program to crash would not be caught during compile-time, unlike in a statically typed language like Java.

2.2 Memory Management

Both Python and Java use automatic garbage collectors, although there are differences in the garbage collection (GC) algorithms used across different versions of these languages.

Although Python is known for reference counts, it may also use a supplemental generation-based garbage collector (accessible by the gc module in the Python standard library) in tandem with reference counting [5]. The advantage of this mixed approach is storage can be immediately and inexpensively cleared when an object's reference count drops to 0, and the generational GC can be used occasionally to detect and free storage pointed at by cyclic references that are not detected by reference counting.

Java, on the other hand, uses a mark-and-sweep garbage collector where older generations are collected concurrently (task threads continue to run during much of the garbage collection process) while newer generations are collected in parallel (all available CPU in the system is used to perform GC as fast as possible) [6][7].

Although the Java garbage collection algorithm is more reliable than Python's reference counting algorithm alone, it is more expensive and performance is unpredictable in real-time programs, which would be a huge downside for a server herd application where updates need to be rapid. Therefore, Python's reference counting GC alongside occasional generational GC is the better choice.

2.3 Multithreading

Multithreading is supported by the Java Virtual Machine (JVM) where true parallelization is achieved with fine-grained locks for shared resources with keywords such as 'synchronized', 'final' and 'volatile'. Python is not thread-safe with its reference counting GC algorithm alone (due to possible memory leaks) and as such, any thread must hold the Global Interpreter Lock (GIL) to access memory space [8]. Therefore, "parallel execution" in Python is really just single-threaded context switching. A proxy herd application would not perform complex computations, which would benefit from multithreading. so a single-threaded program is sufficient and faster since it avoids the unnecessary complexity and overhead of true multithreading. If required, parallelism in Python be achieved with C libraries like Numpy, Pytorch, etc. that offer multiprocessing.

III. *asyncio*

asyncio is an asynchronous concurrent library in Python that uses cooperative multitasking, which is where the scheduler gives exclusive access to a task until it completes or yields control, to achieve concurrency.

3.1 Overview

asyncio facilitates single-threaded and single-process concurrent code with coroutines, which are like subroutines except they can be entered, excited and resumed at many different points (unlike subroutines which can be entered at one and exited at another point) [9]. Coroutines can be suspended and resumed, and the runtime executes another coroutine when a one coroutine is suspended. The *asyncio* package includes the following keywords:

1. async: mark a function as a coroutine.
2. await: suspend current coroutine until awaited function has finished. await must be called within a coroutine on an async function.

The runtime determines which coroutines are to be called through an event loop, where coroutines are scheduled and monitored. Figure 2 presents the working of an *asyncio* event loop.

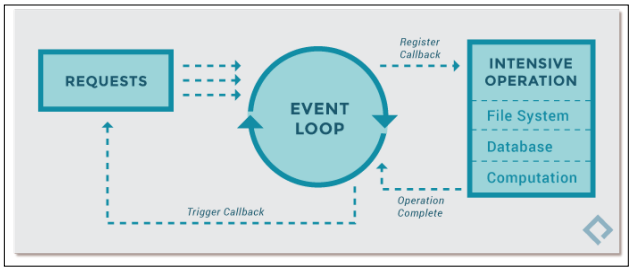

Figure 2: *asyncio* event loop

asyncio is an effective tool in networking applications, where a TCP server must handle multiple TCP clients efficiently while performing I/O operations (reading/writing to network stream).

3.2 Prototype

The *asyncio*-based proxy herd prototype built for this project consists of 5 servers: 'Riley', 'Jaquez', 'Juzang', 'Campbell', and 'Bernard' that are able to "talk" to each other selectively. Every server accepts TCP connections from clients that emulate mobile devices with IP addresses and DNS names. A client may send its location to a server in a message in the following format:

IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1621464827.959498503

Here, the IAMAT command tells the server that the client is at the given latitude and longitude. The server responds to clients with messages following the format:

AT Riley +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1621464827.959498503

Here, Riley is the server ID, followed by the time difference between the client's time stamp and the server's receiving time stamp.

While testing my prototype with the given testing script, I ran into several issues. One that I was not able to solve was the problem of advanced flooding correctness. The advanced flooding correctness score was initially 7/10 when the WHATSAT and IAMAT messages were not being handled by the servers correctly, but it went down to 3/10 once I fixed those. Furthermore, while flooding, I noticed that the servers stopped being able to connect to their neighbors after a few floods, and I got "connect call failed" errors. This leads me to consider that maybe the servers may be taking longer start and end than expected, and that there may need to be longer asyncio.sleep() periods while starting or closing a server.

3.3 Performance

Since *asyncio* was designed with network I/O performance in mind, its performance implications for an application server herd program bode well since voluntary task yielding allows updates to be processed and forwarded to other servers in the herd promptly. However, given Python's lack of support for multithreading, CPU bound operations such as large computations would present a pitfall. *asyncio* is inherently single-threaded so the application would show very poor performance on CPU-intensive requests sent to a server. For more intensive tasks, a multiprocessor framework would be a better choice.

3.4 Newer Features

Updates to *asyncio* in Python 3.9 include:

1. removal of the reuse_address parameter in asyncio.loop.create_datagram_endpoint()
2. addition of a shutdown_default_executor() coroutine, called by asyncio.run()
3. addition of asyncio.PidfdChildWatcher
4. addition of asyncio.to_thread() coroutine
5. When cancelling the task due to a timeout, asyncio.wait_for() will now wait until the cancellation is complete also in the case when timeout is <= 0
6. asyncio now raises TyperError when calling incompatible methods [10]

Update #1 was due to security concerns [11]. Update #4 allowed IO-bound functions to run in a separate thread to avoid blocking the event loop [11], but this can be achieved in older versions by using awaitable loop.run_in_executor(executor, func, *args) which arranges for func to be called in the specified executor. None of the other changes are significant here. Therefore, it is not important to rely on only Python 3.9 or later.

3.5 *asyncio* and Node.js

While *asyncio* is a library, Node.js is an asynchronous event-driven JavaScript runtime environment so the usage of both these tools to run asynchronous I/O programs is quite different. Firstly, Node.js is inherently asynchronous and all the I/O methods in its standard library are asynchronous; these asynchronous versions

are non-blocking and the event loop is able to continue running when one of them is executing [12]. In general, synchronous requests should be avoided while using Node.js since they require the use of additional modules. On the other hand, *asyncio* simply offers asynchronous functionality but is not compatible with synchronous (blocking) Python libraries. The loop.run_in_executor() method can be used with a concurrent.futures.ThreadPoolExecutor to execute blocking code in a separate OS thread (or even process) from the event loop. Otherwise, if a blocking CPU-intensive computation is performed, all concurrent *asyncio* tasks and I/O operations would be blocked [13].

Apart from this, asynchronous behavior is achieved in Node.js and with asyncio in fairly similar ways. Both use async/await keywords, but in Node.js they are built atop the “Promise” object, which represents the eventual completion or failure of an async operation [14]. Additionally, Node.js uses callbacks, which are asynchronous functions that are called at the completion of a given task [15], and these are also present in *asyncio* as subroutine functions that are passed as arguments to be executed at some point in the future.

For the purpose of this report, *asyncio* outperforms Node.js by offering several higher level methods which would make writing code for the application easier, whereas in Node.js these auxiliary functions would have to be manually implemented by the programmer..

IV. Recommendation

Based on the comparisons of Python/Java and *asyncio*/Node.js presented in this report, one may conclude that *asyncio* is a suitable framework for the server herd application in a scenario where there are no blocking (CPU-intensive) tasks or where such tasks are rare enough to not hinder performance. Given that Python does not natively support multithreading, any blocking tasks would either slow down the application or would need to be run in a different thread or process to avoid blocking the event loop - this defeats the purpose of using single-threaded *asyncio* for performance. For an IO task-heavy application, however, Python and the *asyncio* library offer major advantages over the alternatives, namely: easy-to-write code, good performance with quick updates due to event-driven concurrency and cheap garbage collection. Therefore, *asyncio* is suitable for this application.

References

[1] Wikipedia. (2021, June 4). In *Wikipedia*. https://en.wikipedia.org/wiki/Wikipedia

[2] Eggert, P. (2021). *Proxy herd with asyncio*. Retrieved June 4, 2021 from https://web.cs.ucla.edu/classes/spring21/cs131/hw/pr.html

[3] Wikitech. *Wikimedia Infrastructure*. Retrieved June 4, 2021. https://wikitech.wikimedia.org/wiki/Wikimedia_infrastructure

[4] luminousmen. (2020, May 6). *Asynchronous programming. Cooperative multitasking*. *Retrieved June 4, 2021*. https://luminousmen.com/post/asynchronous-programming-cooperative-multitasking

[5] Debrie, A. (2021, May 3). *Python Garbage Collection: What It Is and How It Works* Retrieved June 4, 2021. https://stackify.com/python-garbage-collection/

[6] Javatpoint. *Memory Management in Java*. Retrieved June 4, 2021. https://www.javatpoint.com/memory-management-in-java

[7] Oracle. *Java Garbage Collection Basics*. Retrieved June 4, 2021. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[8] The Python Wiki. *Global Interpeter Lock*. Retrieved June 4, 2021. https://wiki.python.org/moin/GlobalInterpreterLock

[9] Python Software Foundation. *Glossary*. Retrieved June 4, 2021. https://docs.python.org/3/glossary.html#term-coroutine

[10] Python Software Foundation. *What's New in Python 3.9*. Retrieved June 4, 2021.https://docs.python.org/3/whatsnew/3.9.html

[11] Malik, F. (2020, October 4). *10 Awesome Python 3.9 Features*. Retrieved June 4, 2021. https://towardsdatascience.com/10-awesome-python-3-9-features-b8c27f5eba5c

[12] OpenJS Foundation. *Overview of Blocking vs, Non-Blocking*. Retrieved June 4, 2021. https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

[13] Python Software Foundation. *Developing with asyncio*. Retrieved June 4, 2021. https://docs.python.org/3/library/asyncio-dev.html

[14] MDN Web Docs. *Promise*. Retrieved June 4, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[15] OpenJS Foundation. (2011, September 26). *What are callbacks?* Retrieved June 4, 2021. https://nodejs.org/en/knowledge/getting-started/control-flow/what-are-callbacks/

[16] Mondal, A. (2021). *CS131 Week 9* [Discussion Slides]. Retrieved June 4, 2021. https://ccle.ucla.edu/mod/resource/view.php?id=3970188 (not cited in-text)