Suitability and Performance Implications of Server Herd Architecture using asyncio Library

Tian Ye
University of California, Los Angeles

Abstract

In this study we investigate the suitability of implementing a Wikimedia style news system with the additional caveats of needing to facilitate frequent updates through various protocols from highly mobile clients using the Python asyncio asynchronous networking library to implement a server herd architecture. We will also briefly compare the functionality of asyncio in comparison to Node.js and comprehensively compare the performance of Python in comparison to that of Java. From our investigation we conclude that the Python asyncio library is the ideal choice due to having comparative performance to Java whilst maintaining comparable functionality to Node.js.

1 Introduction

There exist a multitude of web stack architectures; the type employed by our example, Wikimedia, is a LAMP-stack architecture that is based on GNU/Linux, Apache, MySQL, and PHP. LAMP uses multiple redundant web servers behind a load-balancing virtual router to achieve maximal reliability and performance. However, for our circumstances, a Wikimedia style news service where updates will be happening far more frequently and clients will be far more mobile, the LAMP-stack architecture will bottleneck at the virtual router and be inflexible in connecting to specific servers.

To combat these challenges, we investigate an application server herd architecture and the suitability of using asyncio library and the Python programming language to implement the server.

2 Suitability of asyncio for Implementation

The asyncio library contains built in support for asynchronous TCP connections; this can be seen through the module's key packages being a pluggable event loop, transport and protocol abstractions, support for TCP, UDP, and SSL, subprocess pipes, and delayed calls. This allows each client conection to the application server to be treated as a TCP connection to an individual server in the herd, and is handled as a coroutine that is added onto the event loop as a callback. The event loop processes a single callback at a time and yields when awaiting input or a response from a server.

The asyncio event loop that runs on the server-side is set up as an asyncio event loop on which asyncio coroutines will run as asyncio Tasks. As an example, when a client connects , the requests will be handled through an asynchronous function defined as the coroutine object and wrapped into an asyncio Task object that is then placed onto the event loop. Per asyncio documentation, the coroutine will be guaranteed to run in the future and will be processed until yield on the event loop until the Task is completed, whereby the server will have provided all its responses to a client's requests[1].

Continuing to examine our implementation of this project from a Python perspective, we see that asyncio is actually an extremely suitable for the implementation of an application server herd structure. Python coroutines are cooperative, non-preemptive multitasking generator functions whose executions are scheduled by a caller such as an event loop. Hence, it follows that coroutines will decide when to yield, and while yielding, they will accept input from a caller.

The logical extension to this functionality is to schedule coroutines corresponding to each client connection as an asyncio Task. The data from the client is passed through an open TCP connection. This will allow servers to accept multiple connections at the same time while maintaining the ability to process input asynchronously. By handling connections as Tasks associated with different coroutines, the reliability of message processing improves as the asyncio implementation guarantees that messages from different clients will not mix in the server's buffer.

asyncio by itself does come with limitations, such as its lack of support for HTTP requests. However, other libraries, such as aiohttp which supports asynchronous HTTP requests, interfaces well with asyncio. aiohttp thus allows us to reliably make GET requests to the Google Places API that is used in the WHATSAT queries in our test servers. Paired with

Python's standard library, we are able to use the json module and format the JSON response from the Google Places API with specific constraints.

One particular issue does stem from asyncio's asynchronous nature. If we were in a synchronous model and sent a valid WHATSAT following a valid IAMAT message, the client could guarantee that the server would have the location stored as the IAMAT message would be processed first. In an asynchronous model, this is not necessarily true as it is possible that the WHATSAT will be processed first and the server would return an error to the client, despite the fact that the client sent the messages in the correct order.

Another issue for asyncio is the fact that each server in the herd model runs the same code. In order to update the code running on the servers, each server must be shut down and restarted for the new code to be updated. This would require client information to be resent as the servers are not expected to retain information when they are shut down. As the number of servers increase in the herd, this issue only compounds. While this issue can be addressed by staggering the restarts and having the servers log and reload client information, this particular limitation is relatively annoying to deal with.

For the most part, asyncio allowed for a relatively painless implementation of the server herd; however, its asynchronous event loop made debugging relatively diifficult, as stepping through the interpreter is relatively useless as the code does not run synchronously, as the name implies. Additionally, asyncio does not offer support for multithreaded implementations. While this can be mitigated by increasing the number of servers in the herd to deal with increased amounts of traffic, if enough of the traffic were being sent to a single server, the server would bottleneck and the problem will resurface. Hence, if the program needs to meet performance benchmarks, asyncio may not be the correct library to use as larger programs will likely require parallelization.

3 Python vs Java in Server Herd

The primary concern for an asyncio implementation of a server herd is that Python's implementation of type checking, memory management, and multithreading may create issues for larger applications. Hence, we will follow with a comparison to the various Java implementations of the language features and initiate a performance comparison. For the purposes of this comparison, we will assume a CPython implementation.

3.1 Garbage Collection

For memory management purposes, Python maintains a private heap that contains all of its objects and data structures[2]. Python's implemenation of garbage colection primarily makes use of reference counts. In a reference count implementation, a given object is stored in memory with an additional field representing the number of times that particular object is referenced in memory. If another object references that object, the reference counter is incremented. When the second object stops referencing the first object, the reference counter is then decremented. This particular method of garbage collection is completely automatic; Python has a set of predefined macros that will sweep the allocated memory and perform the incrementation and decrementation of the reference counts. When an object in the heap has a reference count of 0, Python will garbage collect and free the memory[3].

This however comes at the cost of creating cycles and potential memory leaks. Say we have two objects, x and y at two memory locations, m_1 and m_2, respectively. Let us then have x reference y and y reference x, placing both reference counters at 2. If x and y are each then assigned to different chunks of memory, m_1 and m_2 are no longer referenced by x and y but are still referenced by each other. Hence, their reference counter will never decrement to 0 and their memory locations will never be freed despite the fact that no object resides in that part of memory.

Java on the other hand uses a mark and sweep garbage collection approach as well as using generational garbage collection. At regular intervals, Java will sweep the heap starting at the root blocks of memory, marking any block of memory that is reachable. After the blocks are marked, a second sweep removes any unmarked blocks of memory. As blocks of memory survive sweeps, they increase in generational count, making them less likely to be garbage collected as they are viewed more as persistent blocks of memory. This process generally occurs on a seperate thread from the main program which greatly increases the throughput of Java programs and also makes Java much more memory efficient than Python.

Java achieves true concurrency by operating on a seperate thread and will never result in a memory leak as it does not have the reference count cycle problem faced by Python. Java also avoids the potential pitfalls of race conditions within its garbage collection with its implementations of multithreading, concurrency, and synchronization primitives.

3.2 Multithreading

When it comes to multithreading, Java once again is more efficient than Python. Python's inefficiency stems from its Global Interpreter Lock: a mutex that protects access to Python objects in memory from concurrent access by multiple threads[4]. This means that different threads cannot operate on shared memory and that Python can effectively only achieve interleaving and not true parallelism. This prevents Python from performing concurrent garbage collection which limits its throughput. In the context of of the application server herd, this means that we cannot update the dictionary storing client information at the same time, even if they are two different entries within the dictionary. Hence, Java's advantage over Python is that it permits multiple access and true parallelism.

3.3 Analysis

We see from the sections above that Python fares worse than Java in terms of both memory management and multithreading performance. However, Python makes up for this in its significant advantages in prototyping time and programmer productivity. The fact that Python is dynamically typed results in less mental overhead for the programmer than the static typing of Java. Python is also overall much more concise than Java; in two programs with similar functionality, Python code is generally more readable and compact than its Java counterpart. Furthermore, Python containers are more flexible than Java's and have native support for holding primitives and objects of different types without requiring implementing polymorphic superclasses.

In the context of this server herd application, Python's duck typing only further enhances the benefits of asyncio's asynchronous coroutines. When information is passed from a caller to a callee following a yield, the type of expected object does not need be explicitly defined. This saves the programmer time from researching which object types should be expected without sacrificing the benefit of Python's strong typing.

4 asyncio vs Node.js in Server Herd

Node.js is an event-driven, asynchronous, non-blocking I/O model that uses an event loop. These features are also associated with asyncio. Similarly to asyncio, Node.js schedules functions as callbacks onto the event loop and are processed until they yield, whereby more work on the event loops is processed as the yielded callback function awaits further input[5]. Essentially, both asyncio and Node.js have the same methodologies; they are analogous to one another.

Compared to Node.js, asyncio is much more established as will well experimented with. Node.js on the other hand is newer and has the advantage of having a better maintained library with constant open source updates to meet the newest web standards. While Python is portable to web and mobile applications, it was initially developed as a scripting language and lacks the development community that Node.js has. Node.js on the other hand is built on the V8 web engine which is designed to build fast, lightweight, and scalable dynamic web applications. Furthermore, Node.js is preferred in developing web applications as the frontend and backend for a given application will be in the same language, resulting in fewer incompatabilites and causing the application to run seamlessly. This can be seen in Node.js's latency advantage over asyncio.

However, Node.js is not as suitable for processor intensive applications; in the case of the application server herd which requires the ability to handle large throughput of connections efficiently, the asyncio library is a better fit.

5 Conclusion

From the investigation above, we can say that while Python's asyncio is not perfect, both the language and module are suitable for the task at hand and do not present any massive performance downsides for the task at hand in comparison to alternative language implementations. Hence, our study recommends the use of asyncio for the implementation of a Wikimedia-style application server herd for the purpose of a news service.

References

[1] "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*, Python Software Foundation, docs.python.org/3/library/asyncio.html.

[2] "Memory Management." *16.2. Threading - Higher-Level Threading Interface - Python 2.7.15 Documentation*, Python Software Foundation, docs.python.org/3/c-api/memory.html.

[3] Artem Golubin. "Garbage Collection in Python: Things You Need to Know." *Artem Golubin*, Artem Golubin, 7 Jan. 2019, rushter.com/blog/python-garbage-collector/.

[4] Wouters, Thomas. "Global Interpreter Lock." *Global Interpreter Lock*, Python Software Foundation, wiki.python.org/moin/GlobalInterpreterLock.

[5] "Python vs Node.js: Which Is Better for Your Project." *7 Best Social Network Software. How Much Does It Cost to Build Own?* | DA-14, da-14.com/blog/python-vs-nodejs-which-better-your-project.