Performance Implications and Suitability of asyncio Implementation of Application Server Herd Architecture

Austin Guo – University of California, Los Angeles

Abstract

In our study we investigate the performance and suitability of the Python asyncio asynchronous networking library for implementation of an application server herd architecture in a Wikimedia-style news system that requires accommodation of quick and frequent updates through various protocols from clients that are highly mobile, in addition to the maintainability of such a Python implementation. We also frame a brief analysis of the functionality and approach of asyncio in comparison to Node.js, as well as a more comprehensive presentation of the performance of Python in comparison with Java. As a result of our investigation we recommend use of Python and the asyncio library in prototyping and implementing the application server herd architecture in the Wikimedia-style news system in question, due to many ease of access and maintenance reasons, as well as comparative performance to Java and comparative functionality to Node.js.

Introduction

There exist many different web stack architectures, each having different best use cases depending on implementation philosophy and tradeoffs made from design choices. For example, Wikimedia and its related sites follow a LAMP-stack architecture based on GNU/Linux, Apache, MySQL, and PHP, which take advantage of using multiple redundant web servers behind a load-balancing virtual router for reliability and performance. However, for a Wikimedia-style service designed for news, where updates to articles occur far more often, connections must be received by various different protocols, and clients tend to be more mobile and may find benefit in connecting to closer proximity servers as they move, the LAMP-stack architecture employed by Wikipedia faces a bottleneck in the load-balancing router and inflexibility in connecting to specific servers.

In order to work around this bottleneck, we investigate an application server herd architecture and analyze the suitability of using the asyncio asynchronous networking library as well as the Python programming language to implement such an architecture.

Our investigation provides conclusive results that attest to the ease of access to asyncio in terms of being able to easily produce and maintain asyncio-based code to implement the application server herd architecture. Further, we show that an asyncio-based implementation of the application server herd architecture eliminates the bottleneck associated with the load-balancing router by allowing client connections to any server and accommodates large numbers of client connections through asynchronously handling client connections as asyncio-supported tasks. Meanwhile, asyncio still allows for easy preservation of reliability, as the asynchronous connection framework supports server connections to servers such that client information can flood to each server in a performant manner. However, asyncio does present minor inconveniences for our use case, for example lack of built-in support for TCP and SSL, though we may compensate by using external libraries that interface well with asyncio such as aiohttp.

As a result of our study we conclude that while imperfect, the Python asyncio library provides almost comprehensive support for our application server herd system, while also providing the extra convenience of Python as a development language, reducing development time by providing convenient features like duck typing and automatic memory management. As a result, our recommendation is in favor of using asyncio and Python in the implementation of an application server herd service.

1. Implementation Design

In this study's design of the application server herd we are able to leverage co-routines and event loops to accept connections asynchronously and process input and output in a non-blocking manner.

Each client connection to the application server herd is treated as a TCP connection to a specific server in the herd, and is handled as a co-routine that gets added onto the event loop as a callback. The event loop processes one callback on its queue at a time, and yields when awaiting input or a response from a server function processing a request.

When a client connects, the server expects the client to send a message that follows one of three formats: the message must be a IAMAT message, a WHATSAT request, or an AT message.

1.1. IAMAT Command

The IAMAT message serves to inform the application server herd of the current location of a certain client. The IAMAT message format consists of four whitespace separated fields, and may be represented by the expression: *IAMAT <client ID> <ISO 6709 latitude and longitude> <POSIX time when message sent>*, where all fields must be non-whitespace strings. Server responses sent back to the client will be of the form: *AT <server ID> <clock skew time difference> <client ID> <ISO 6709 latitude and longitude> <POSIX time when message sent>*, where all fields are also non-whitespace strings and the clock skew time difference represents the difference between the POSIX time that the client originally sent the message and when the server received the message.

1.2 WHATSAT Command

The WHATSAT message serves to provide a response to the client informing of the places nearby their current location. The WHATSAT message format consists of four whitespace separate fields, and may be represented by the expression: *WHATSAT <client ID> <radius> <number of results>*, where radius must be a value in the range [0, 50] and the number of results is the number of Google Places results to return to the client within the radius of the client's location, where number of results must be a value in the range [0, 20]. The response format is the same as the IAMAT response, except after a newline the JSON response from Google Places associated with the query formed by information provided by the WHATSAT request is appended at the end, limited to the number of results the WHATSAT request specified. An exception to this response format, however, is when the server does not have information on the client's location, meaning the client has not previously provided its information through an IAMAT message to the server herd. In this case, the server will respond with an invalid message response, of the format: *? <message>*, where message is the message the client sent.

1.3 AT Command

The AT message serves as a method by which servers propagate messages to each other within the application server herd. The format is the same as the server responses to clients for an IAMAT message except with an extra field denoting which server the message came from.

1.4 Malformed Requests

In any scenario, if the client provides a malformed query to any server in the application server herd, the server will respond with an invalid message response, of the format: *? <message>*, where message is the message the client sent. Malformed queries may mean the message contains an invalid command (one that is not IAMAT, WHATSAT, or AT), or one of the non-whitespace fields in the message is incorrect. A comprehensive list of invalid requests conditions include: if the latitude and longitude information is not provided in ISO 6709 format in an IAMAT request, if the timestamp of message dispatch is not in POSIX UTC time, if there are an incorrect number of whitespace separated fields, if a WHATSAT radius or number of results field is not an integer in the range previously specified, or if any field of any command has whitespace characters.

1.5 Message Design in Client Communication and Server Flood

This message design serves as the basis for how the Wikimedia-style news service will communicate with clients and keep the latest information on each server within the herd. Clients can provide and request information through TCP connections using IAMAT and WHATSAT, while servers propagate IAMAT information to other servers within the herd by means of a flooding algorithm, which has a server flood AT messages to its designated partner servers (which it is responsible for informing and receiving information at any time), unless the designated server is the original source of the flood or is the server propagating information to the current server. In both cases, the server in question already has the information being propagated and does not need to be re-informed.

It is important to note that in server to server connections, the server connecting is received as a client connection by the other server, which only realizes a server connection has occurred when it receives an AT message. This has the advantage of greatly simplifying the design of our server prototype.

2. Suitability of asyncio for Implementation

The asyncio library provides built-in support for asynchronous TCP connection handling: in fact, the module's key package contents are a pluggable event loop, transport and protocol abstractions, concrete support for TCP, UDP, SSL, subprocess pipes and delayed calls, and various co-routines and tasks based on "yield from"[1], as our design specified. These contents will greatly facilitate the implementation of the application server herd infrastructure and make

asyncio an extremely suitable library to use to run and exploit server herds.

2.1 Asyncio Event Loop, Co-routines and Tasks

The module allows us to begin developing server-side code by setting up the asyncio event loop, on which asyncio co-routines will run as asyncio Tasks. For example, when a client connects, we will handle the client's request(s) throughout the connection by using an asynchronous function defined as a co-routine object and wrap it in an asyncio Task object that will be placed on the event loop. According to the asyncio documentation, the co-routine will then be guaranteed to run in the future, and will be processed until yield on the event loop multiple times until completion of the Task, at which point the server will have provided all its responses to the client's requests and the client connection will have ended[1].

More precisely, implementing the application server herd infrastructure using the asyncio module is an extremely suitable application of asyncio's functionality and as a result, has numerous positive implications with respect to performance and speed of prototyping. As described previously, asyncio's offered package contents align almost exactly with our desired functionality. To better understand the extent of the truth behind this statement, we must examine Python co-routines.

Python co-routines are cooperative, non-preemptive multitasking generator functions whose executions are scheduled by a caller such as an event loop. Therefore, it follows that co-routines decide when they yield, and at the point of yielding, they will accept input from the caller in a manner very similar to inter-process communication – with the exception that the communication is internal to a process (this means the information trade is very much a trade across function stack frames)[2].

As a result, it is possible for the asyncio event loop to schedule co-routines corresponding to client connections as asyncio Tasks, while passing in input data from the client through an open TCP connection. In this manner, asyncio will allow the servers to accept multiple connections at the same time while maintaining the ability to process input asynchronously. In the same thread, this manner of handling connections as Tasks associated with co-routines also allows for servers to maintain separate buffers of messages for each client connection. This improves the reliability of the message processing in our asyncio implementation by guaranteeing that messages from different clients will not mix to form garbage values in our server's buffer.

2.2 Other Components

Other components of the implementation also interface with asyncio. Because asyncio does not support HTTP requests, in order to implement the WHATSAT command handler we must make use of the aiohttp library, an asynchronous networking library that supports asynchronous HTTP requests and interfaces well with existing asyncio implementations. The aiohttp module will allow us to reliably make GET requests to the Google Places API formed from WHATSAT queries, and paired with the json module in Python's standard library, will allow the server to reformat the JSON response from Google Places to have only the specified *<number of results>* results displayed and therefore respond to such queries with a correctly formatted response.

Further, we may use Python efficient built-in data types to assist with our implementation data structures. In order to keep track of each client's most recent locations, the timestamps of receipt and transfer, and the original server communicated to, each server can maintain a cache implemented using Python dictionaries. The result is not only high performance due to its Linked Hashmap implementation (O(1) access and updation), but also ease of access in terms of prototyping speed and readability. This drastically reduces the time necessary for a given server to respond to a WHATSAT command, and also allows a server to quickly update its information on a client when receiving an IAMAT or AT command. Similarly, we are able to maintain a dictionary to represent the digraph of communications between servers, and various lists and regex objects to assist with field validation and command processing.

3. Analysis of Python vs Java in Server Herd

A major concern in our investigation of an asyncio implementation of a server herd prototype is that Python's implementation of type checking, memory management, and multithreading may cause problems for larger applications. As a result, we follow with a discussion on Java's implementation of such language features and compare their performance. For the duration of our discussion, we will assume a CPython implementation.

In order to better understand the differences in Python and Java's implementation of garbage collection, we must first understand the concept of reference counts, generational GC and concurrent mark and sweep.

Python's implementation of garbage collection makes extensive use of reference counts. In a reference count

implementation, each object is stored in memory with an additional field for keeping track of how many times the object is referenced in memory. Any time another object references the current object's memory, the reference count will be incremented. Whenever this second object does not reference the current object's memory anymore, the reference count is decremented.

The advantage of having such a method of garbage collection is that it is completely automatic. Python has a set of predefined macros that will sweep over allocated memory and automatically increment and decrement reference counts when objects in memory are being referenced by other allocated objects. Further, at regular intervals, Python will garbage collect and perform a sweep over heap memory to free any objects with reference counts of 0.

However, reference counts have the disadvantage of allowing cycles. For example, it is possible for object A at memory M1 to reference object B at memory M2 and object B to reference object A, placing both objects at reference count 2. Now we assign object A to a different chunk of memory M3, and assign object B to another different chunk of memory M4. In this scenario, M1 and M2 are now not referenced by object A and B, but are still referenced by each other. Thus, the reference counts for M1 and M2 only decrement to reference count 1, and are never freed even though no object resides at those memory locations any more.

Java, on the other hand, uses a more traditional method of mark-and-sweep garbage collection in addition to generational garbage collection. This means at regular intervals, Java will sweep the heap starting at root blocks of memory and mark every block of memory in the heap that can be reached from these root blocks. After all such blocks are marked, a second sweep over the heap allows Java to remove non-marked blocks of memory. As blocks of memory survive waves of garbage collection, they increase in generation, which makes them less and less likely to be garbage collected as they are viewed as more persistent blocks of memory. Though this sounds like an extremely slow process, requiring two passes over the entire heap, often Java's garbage collection occurs on a separate thread, greatly increasing the throughput of Java programs and making Java much more memory efficient.

From our previous discussion of the mark-and-sweep and reference count models of memory management, Java memory management is much more effective than Python's. Not only does it achieve true concurrency by operating on an entire separate thread, it also will never leave unused memory on the heap unlike Python with its reference count cycle problem. Further, though it may appear that there will be race conditions with Java clearing memory as the program is running, Java's implementations of multithreading and concurrency have efficient built-in synchronization primitives that ensure reliability.

On the topic of multithreading, Java has a distinct advantage over Python. A significant reason for Python's inefficient multithreading is the Global Interpreter Lock. The Global Interpreter Lock is a mutex that protects access to Python objects in memory from concurrent access by multiple threads[6]. This is a considerable downside of Python, as it means different threads cannot operate on shared memory, and effectively a multithreaded Python program can only achieve interleaving and not true parallelism. This not only affects the ability of Python to perform concurrent garbage collection to increase throughput, but it also affects our ability to implement our application server herd in a multithreaded manner: when handling multiple client connections, the dictionary storing client information will not be able to be updated on two different threads at the same time, even if two different buckets in the dictionary are being updated. As a result, Java's multithreading is more effective, allowing multiple accesses to the same object and true parallelism.

Though Python lags behind Java in general performance in terms of memory management multithreading, Python has significant advantages in terms of prototyping speed and programmer productivity. The widely accepted increase in application performance in terms of developer productivity when working in Python rather than Java is 5-10 times productivity boost[4]. This figure may be largely due to Python's conciseness and compactness compared to Java. In implementing similar functionality, Python code is generally more compact and more readable, though in large programs with many interacting modules Java's static typing may make it much easier to trace data being passed around across class files. However, in general, Python's containers are more flexible and have native support for holding primitives and objects of different type without extra implementation of polymorphic superclasses, and code is more concise in Python.

Further, Python's use of duck typing compounds the benefits of asyncio's use of asynchronous co-routines; as previously noted, when information is passed from caller to callee following a yield, the developer does not have to explicitly define the type of the object expected from the caller. This sort of type-inferencing

based on context in Python saves the developer time and research effort into which range of objects should be expected when prototyping, without sacrificing the benefit of Python's strong typing, which often tends to be more reliable and performant than weak typing[5].

4. Problems with Implementation

For the most part, asyncio allowed for relatively painless implementation of a server herd. However, dealing with any asynchronous event loop development will make it more difficult to debug, since the code does not run synchronously and thus stepping through the interpreter is relatively useless. Debugging involved using lots of console log statements and was considerably less convenient than using a synchronous implementation without asyncio.

Further, asyncio does not offer support for multithreaded implementations. In order to make our program more parallelizable, as a more large scale program may need to meet performance benchmarks, asyncio may not be the correct library to use[2].

5. asyncio vs Node.js

Implementations of the asyncio library and the Node.js framework are strikingly similar. Node.js is an event-driven, asynchronous and non-blocking I/O model, and uses an event loop just as asyncio does. Similarly, functions are scheduled as callbacks onto the Node.js event loop, and are processed until they yield, at which point more work on the event loop is processed as the yielded callback function is awaiting input[3].

These characteristics of Node.js are inherently analogous to the Python asyncio characteristics described earlier. Explicitly, Node.js callbacks are essentially asyncio co-routines that yield execution and continue execution later when rescheduled by the event loop, and the Node.js event loop can be directly compared to the asyncio event loop[3]. Both implementations string together chains of callbacks to get work done.

In other avenues, asyncio is a much more established an well-experimented library. On the other hand, Node.js is much newer, but as a result has the advantage of being a much better maintained library with constant open source updates to meet the newest web standards. Further, Node.js was built on the V8 web engine for building fast, lightweight and scalable dynamic web applications. Though Python has the advantage of being extremely portable to web and mobile applications, it was initially developed as a scripting language and does not have the immense web development community Node.js has supporting and maintaining it[3]. On top of this, Node.js is also preferred in developing web applications because the frontend and backend for given applications will necessarily be in the same language, and thus fewer language incompatibilities will be encountered and the application will run much more seamlessly. This has numerous positive benefits for companies developing server-side and client-side code for applications, and can be clearly seen in Node.js's latency advantage over Python asyncio[3].

However, Node.js is not as suitable for processor intensive applications. In the case of the application server herd, which must be able to handle large throughput of connections efficiently, the asyncio library may see better fit.

Conclusion

From our investigation, we see that though Python and asyncio are not perfect, the language and module both are extremely suitable for the task at hand and do not present significant performance downsides compared to alternative implementations. As a result, our study recommends use of asyncio and Python to implement the Wikimedia-style application server herd service.

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*, Python Software Foundation, 10 Mar. 2018, docs.python.org/3/library/asyncio.html.

[2] Flaxman, Michael. "Python 3's Killer Feature: Asyncio." *PAXOS*, 21 June 2017, eng.paxos.com/python-3s-killer-feature-asyncio.

[3] Saba, Sahand. "Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js." *Math Code by Sahand Saba Full Atom*, 10 Oct. 2014, sahandsaba.com/understanding-asyncio-node-js-python-3-4.html.

[4] Sehar, Uroosa. "Java vs. Python: Which One Is Best for You?" *Application Performance Monitoring Blog*, AppDynamics, 21 Mar. 2017, blog.appdynamics.com/engineering/java-vs-python-which-one-is-best-for-you/.

[5] Sommers, Bill Venners with Frank. *Strong versus Weak Typing*, Artima, 10 Feb. 2003, www.artima.com/intv/strongweak.html/.

[6] Wouters, Thomas. *Global Interpreter Lock*, Python Software Foundation, 02 Aug. 2017, https://wiki.python.org/moin/GlobalInterpreterLock