Building Application Server Herds with Python's Asyncio Library

Abstract

I investigate the ability of Python's Asyncio library to create maintainable, reliable, and efficient server herd applications by creating a parallelizable proxy for Google Places API. I then compare the reliability, effectiveness, and functionality of Python server herd applications to Java and Node.js applications. In particular, I compare Python and Java's type checking, memory management, and multithreading characteristics. Additionally, I compare the functionality of Python's Asyncio library to a Node.js approach. I conclude that Python's Asyncio library is well suited for exploiting the benefits of application server herds due to its maintainability, ease of programming, and asynchronous parallelizability.

1. Introduction

Wikipedia uses a LAMP platform running on multiple, redundant webservers behind a load-balancing router. However, this architecture makes it difficult to add new servers, and the application server can become a bottleneck. In order to build a Wikimedia news style service supporting frequent article updates, various protocols, and more mobile clients, an application server herd architecture may provide better performance. In this architecture, multiple application servers could serve requests and communicate information between one another while minimizing accesses to centralized databases. This architecture could make it easier to add new servers to cover more mobile users, and it could improve throughput by minimizing bottlenecks.

Python's Asyncio asynchronous networking library seems like a good fit for application server herds, because its event-driven approach should allow messages to be propagated rapidly across the herd. In order to determine whether the Asyncio library and Python language system are a good choice for this application, I use the Asyncio library to build a simple proxy server herd, and I evaluate its effectiveness. I also analyze the reliability and flexibility of Python applications in comparison to Java applications, and I compare the functionality of Asyncio to Node.js. As a result of my analysis, I conclude that the Asyncio library should be used for developing the Wikimedia application server herd.

2. The Asyncio Python Library

The Asyncio Python Library provides a foundation for asynchronous frameworks. At a low level, Asyncio uses an event loop that runs in a thread to schedule asynchronous tasks and coroutines[1]. When a task voluntarily yields with an await statement, the event loop can begin executing another task. This is particularly useful for blocking system calls, I/O, and other events that would normally force the process to wait until the blocking event has completed. If another task has been created, then an asynchronous blocking event such as reading a message over a TCP socket can be initiated while the event loop executes the other task. After the blocking event completes, the event loop will eventually return control back to the original task that initiated the blocking event.

2.1. Asyncio Networking Library

The Asyncio library provides a high-level API for building asynchronous servers. As of Python 3.7.2, Asyncio provides an asynchronous function call, "start_server" that takes a callback accepting a reader and writer, an IP address, and a port number. The server can then be run with the asynchronous function "serve_forever." This API only requires the developer to implement an async callback, which can read incoming data from the reader class and write data back to clients with the writer class[1]. The simplicity of this design, which can be implemented in about 10 lines of code, provides astounding effectiveness. The Asyncio library handles all incoming connections and schedules each callback as an asynchronous task. This further simplifies the programmer's responsibilities as multiple connections between the server and its clients can be handled simultaneously without any extra burden on the developer. Clients can be created with similar ease, and there are libraries like aiohttp that greatly simplify the process of creating and handling http requests with asynchronous support from the Asyncio library.

2.2. Asyncio Pros and Cons

The Asyncio library demonstrates many strengths. Its simplicity and high-level APIs allow programmers to quickly develop efficient networking applications. These asynchronous applications are particularly effective for application server herds, because many simultaneous connections can be handled with ease, and blocking events will allow the event loop to transfer control to another task. For example, if a server needs to communicate with a central database with a large latency, its request can be created as an asynchronous event so that the server will not be blocked while waiting for a response. In the meantime, the server can handle other client's requests.

In addition, the Asyncio library has many advantages over Python's threading library. By using an event loop, it reduces the overhead incurred by spawning and maintaining multiple threads. Furthermore, while Python's threading library may require each thread to be scheduled before it can determine if the thread is blocked, the Asyncio event loop will not schedule tasks until the blocking operation completes[5]. Asyncio also gives developers much more control over managing the places at which task switches can occur. This allows developers to largely avoid synchronization concerns, nasty data races, and dead lock conditions.

Despite these benefits, Asyncio is not without its disadvantages. The main disadvantage of Asyncio is that it primarily runs on a single thread[1]. This prevents the Asyncio library from utilizing multiple cores in parallel. Unfortunately, Python's Global Interpreter Lock prevents any Python program from truly running in parallel[2]. This contrasts with Java and is explained in depth later (see Section 4.3).

3. Google Places Proxy Implementation

My Python application is designed to be run on an application server herd of five servers and implements a proxy for Google Places API services. It allows for clients to send their location to any server and request information from Google Places about the nearby places of other clients. Due to the functionality of the Asyncio library, each connection to a server is created as an asynchronous task that can yield to other tasks or client connections when blocking I/O or other asynchronous functions are awaited. For example, if a request to Google Places has a long delay, other clients can be served while the request is pending, which increases throughput.

Additionally, each server propagates updated client information to its neighboring servers on the network through TCP connections. Servers only open connections with one another when sending updates, which simplifies handling downed servers. If a server goes down, the herd will not be able to open connections with that server, but they will begin to talk to it again as soon as it comes back online. Client's communicate to servers through TCP connections by sending IAMAT and WHATSAT messages.

3.1. IAMAT

Clients send their location information to servers with IAMAT commands. These commands have the format:

IAMAT <client name> <ISO 6709 client location> <POSIX client time>

If a message is invalid, the server will reply with "?" followed by the invalid message. Otherwise, it replies with the format:

AT <server name> <time difference> <client name> <ISO 6709 client location> <POSIX client time>

The time difference field indicates the difference in time between the time when the server processed the message and the POSIX time sent by the client. The time difference field can be negative due to clock skew. If the time field in the client's message is more recent than the time field of the same client in the server's cache, the response message is saved under the client's name in the server's cache so that WHATSAT messages will look up the most recent location of a client. A flood message is then sent out to the server's neighbors so that they will receive the updated location of the client.

3.2. WHATSAT

Clients can send WHATSAT messages to a server to search for places nearby another client with Google Places. WHATSAT commands have the format:

WHATSAT <client name> <radius> <result limit>

This command specifies a client in the server's database to use as the origin of search for places within the number of kilometers specified by the radius argument. It also limits the number of places returned based on the result limit argument. Servers look up the location of the target client in their cache and forward the search results from Google Places to the requesting client. The target client must exist in the server's cache, the radius must be 50 kilometers or less, and the result limit must be 20 or less. Servers respond to invalid messages with a "?" followed by the invalid message. Servers respond to valid WHATSAT messages with:

AT <server name> <time difference> <client name> <ISO 6709 client location> <POSIX client time> <search results>

The server name, time difference, client name, ISO 6709 client location, and POSIX client time are identical to the database fields created by the server that received the IAMAT message from the target client. The search result is a JSON formatted message containing the results returned by Google Places API.

3.3. Flood Messages

Servers exchange information regarding their clients through a flooding algorithm. Whenever a server updates its database for a client due to an IAMAT command from a client or an AT message from another server, it sends an AT message to all its neighbors. The flood messages have the format:

AT <original server> <time difference> <client name> <ISO 6709 client location> <POSIX client time> <sending server>

The first five fields are created by the server that first receives the IAMAT message, so all servers will know the server from which the flood is originating as well as the client's new data. Servers will not send the flood message back to the originating server or to the server from which the message was last sent (indicated by the sending server field). This prevents an infinite loop of flood messages. By exchanging client information in this manner, all servers can be queried for WHATSAT responses for any client, and if a server goes down, the client's information is not lost.

3.4. Demonstration Results

The application server herd proxy for Google Places was surprisingly simple to implement and appears to work well for its intended application. The capability for any server to handle any WHATSAT requests demonstrates the ability to distribute client requests among different servers in the application server herd without losing information, which improves reliability and performance. Another benefit of the Python application is that more servers could easily be added by adjusting only a few data structures.

In addition, the server logs show that a server often receives multiple messages before sending a response to each client, which demonstrates that many client requests are handled simultaneously. The asynchronous design improves the throughput of each server, particularly for WHATSAT requests that require communicating with Google Places, and the logs show that many requests are completed per second. Finally, the application is relatively reliable, maintainable and extendable since there are no data races or synchronization issues.

4. Python vs Java

Java is well designed for many server applications due to its cross-platform functionality through the Java Virtual Machine and built-in support for multithreading and concurrency. Like Java, Python is an interpreted language that can easily be deployed on different computer architectures. However, it differs significantly from Java in its type checking, memory management, and multithreading capabilities.

4.1. Python vs Java: Type Checking

Python is a dynamically typed language, which means that type errors are not detected until runtime. This can speed up development because programmers do not need to explicitly define every variable and function return type. It also allows for more flexible development because the same function can be used to return different types, and function arguments will accept any type without the need for generics and complicated subtype relations.

Python's dynamic type checking often reflects duck typing. Rather than checking for types, duck typing simply checks for the presence of methods[3]. This also provides developers with increased flexibility because methods such as "len()" and "str()" can be applied to any object as long as it defines that method. While dynamic type checking makes it easy for programmers to quickly develop concise and powerful functionality, it can be dangerous. Runtime errors could be problematic in an application server herd, and dynamic type checking makes it difficult to ensure that there will not be type errors at runtime.

In contrast, Java uses static type checking, which means that its types are checked before it is translated to bytecode. This has the benefit of preventing runtime type errors and improves the reliability of the program. It can also improve performance by reducing runtime checks and allowing for more efficient optimization techniques. However, it also places a burden on programmers to explicitly define types. While generics and subtype relations can be used to improve flexibility, static type checking makes it much more difficult to write programs that are as flexible as those written in a dynamically typed language.

While Java's static type checking provides better reliability and performance, Python's dynamic type checking allows for far simpler program development and greater flexibility. Python also supports type annotations, which can be used for programmers to communicate intended types and improves maintainability[3].

4.2. Python vs Java: Memory Management

Python and Java both provide garbage collectors, which improves reliability and simplifies development by eliminating dangling pointer exceptions and memory leaks. However, Python and Java differ in their implementation of memory allocation and garbage collectors.

Python allocates memory in large chunks called arenas and internally handles all memory accesses[7]. When a new object is created, it is placed in an arena within a pool of objects of similar size, and its reference count is set to 1. Whenever another reference to the object is created, its reference count is incremented, and whenever a reference to the object is removed, its count is decremented. If the count reaches zero, then the object can no longer be reached by the program and can safely be deallocated. The object is then placed on a free list within its pool so that its space can be reused.

This reference counting mechanism incurs some extra overhead every time a variable is assigned, but it immediately frees garbage and avoids the downtime of mark and sweep mechanisms. In addition, Python's use of pools of similar sized objects, or quick lists, allows for efficient space management. By placing all objects of a similar size in the same pool, fragmentation is reduced because the size of all free blocks in the pool will be similar to the space needed for new objects of the same type (and similar size)[7].

However, reference counting fails when circular references are created. If O1 refers to O2, O2 to O3, and O3 to O1, then the reference count of each object will never reach zero, so these objects can never be freed by reference counting alone. In order to avoid memory leaks in these cases, Python includes a backup mark and sweep garbage collector in case circular references are used.

Java traditionally uses free lists and a mark and sweep garbage collector. This garbage collector will occasionally traverse through all references to memory and mark the objects that are reachable. The sweep phase will then return any unmarked object back to the free list. This garbage collecting mechanism avoids all memory leaks, even for circular references, but it can bog down the program while it is running. In practice, the garbage collector typically runs on a separate thread which reduces the overall performance impact on the program[8].

Modern versions of Java use more advanced memory management schemes in which objects are stored in generations and typically only the youngest objects are garbage collected. This can further improve the performance of garbage collection and can also be used to reduce cache pollution.

Overall, Python and Java both provide efficient memory management mechanisms that handle the gory details of allocating and deallocating memory, which makes their applications more reliable and maintainable.

4.3. Python vs Java: Multithreading

Python's memory management mechanism of reference counting poses constraints on its multithreading capabilities. Because the reference count for each object in memory is shared between threads, updates to this count create data races between threads. In order to avoid these data races, Python uses a Global Interpreter Lock to ensure that only one thread can ever execute bytecode at a time[4]. This prevents Python from running threads on multiple cores in parallel, as only one thread can use the interpreter at any one time.

If threads are CPU-bound and do not need to do I/O or other blocking calls, then the locking overhead from running multiple threads results in performance that is worse than single threaded applications[2]. However, I/O-bound Python programs still benefit from multiple threads as a blocked thread will not stop the execution of the entire program[4]. In addition, Python supports multi-process applications, which allows a program to spawn multiple processes to run in parallel on separate cores. Unfortunately, multi-process programs generally take up more overhead, are less efficient, and cannot communicate as directly as multi-threaded programs.

Java was designed with multi-threading in mind, and it provides built-in support for running multiple threads and attempts to simplify synchronization through techniques such as monitor locking. In contrast to Python, Java threads can be run in parallel on separate cores, which allows CPU-bound applications to achieve far superior performance than similar Python applications.

For CPU-bound applications, multithreaded Java applications will far outperform their Python counterparts. However, multithreaded Java and Python programs will likely achieve a more similar performance for I/O bound applications.

4.4. Python Vs Java: Conclusion

Although Python and Java can both be used to produce high performing server applications, Python's Asyncio library is well suited for networking applications that perform a lot of blocking I/O events (like application server herds). In this context, threads will mostly be I/O bound, so Java's capability for parallel multithreading will not provide significant performance gains. As a result, an asynchronous event driven approach is simpler and may achieve similar performance to multithreading.

In addition, Python's reliable memory management system will allow applications to run for extended periods of time without leaking memory or crashing due to illegal memory accesses. Finally, Python's dynamic type checking allows for simple, quick, and flexible program development. Although testing will be needed to ensure reliability, this type system allows for complex interactions with libraries like Asyncio that would be cumbersome with static typing. Since the Wikimedia application will mostly be performing article updates and serving news articles to clients, the application will be I/O bound. As a result, Python is better suited than Java for the application server herd under consideration.

5. Python vs Node.js

Python with Asyncio and Node.js exhibit many similarities. Both use event loops and asynchronous operations with callbacks to avoid blocking. This allows both Python and Node.js to build frameworks for scalable networking server applications that can handle multiple client connections simultaneously.

While the Asyncio library was developed to provide asynchronous support to an already existing language, Node.js was developed specifically for asynchronous, event driven server applications. This leads to some differences in the language systems. Node.js uses JavaScript and is built on the modern V8 web engine[6]. While the event loop in Python is provided through the Asyncio library and must be executed via a blocking call, Node.js creates and executes the event loop as soon as the input script is executed. The event loop in Node.js runs during the entire duration of the program, so almost every function in Node.js is an asynchronous event that cannot block the process[6]. In addition, while Python supports multithreading but not truly parallel multithreading, Node.js does not even include multithreading functionality.

Python with Asyncio and Node.js could both likely be used to create an effective application server herd for the task at hand, so the choice largely depends on the preference for Python vs JavaScript. One advantage of Python's Asyncio library over Node.js; however, is that Asyncio has been developed over a longer period of time and has been more widely used and tested. In addition, Python has a larger and more varied set of supporting libraries and high-level APIs than JavaScript and Node.js.

6. Conclusion

Python with the Asyncio Library makes it relatively easy to write efficient and maintainable application server herds and would be a good choice for implementing the Wikimedia application. Although testing and type annotations will be important to ensure that Python's dynamic type checking does not lead to runtime errors, dynamic type checking will allow for flexible and extensible development of the application. In addition, Python's memory management system will prevent memory access exceptions and memory leaks, allowing the servers to run uninterrupted for extended periods of time. While other languages like Java have much better support for parallel multithreading, Python's Asyncio library is very well designed for the blocking network operations that the server will mostly be handling.

As demonstrated in the demo application server herd, a relatively small amount of code (about 200 lines including comments), can implement efficient asynchronous servers. The development was surprisingly simple due to dynamic type checking and Asyncio's high-level APIs. Despite its simplicity, the sample application server herd can effectively handle simultaneous connections from many clients, communicate with a centralized database, and flood inter-server messages to the rest of the herd. While simple, this demo highlights the simplicity and effectiveness of developing with Python and Asyncio.

Perhaps the most compelling benefit of Asyncio is its simplicity, which will allow developers to more easily verify correctness, make changes, and extend the application. Asyncio therefore makes it easier to develop a maintainable and effective application than other languages like Java. As a result, I recommend using Python and Asyncio for developing the Wikimedia application.

7. References

[1] *Asyncio Python Library*, https://docs.python.org/3/library/asyncio.html

[2] Beazley, David, *Understanding the Python GIL*, http://www.dabeaz.com/python/UnderstandingGIL.pdf

[3] Hjelle, Geir Arne, *The Ultimate Guide to Python Type Checking*, https://realpython.com/python-type-checking/

[4] Humrich, Nick, *Asynchronous Python*, https://hackernoon.com/asynchronous-python-45df84b82434

[5] Mansun, Abu Ashraf, *Async Python: The Different Forms of Concurrency*, http://masnun.rocks/2016/10/06/async-python-the-different-forms-of-concurrency/

[6] *Node.js*, https://nodejs.org/en/

[7] Vantol, Alexander, *Memory Management in Python*, https://realpython.com/python-memory-management/

[8] Worthington, John, *What is Garbage Collection in Java and Why is it Important*, https://www.eginnovations.com/blog/what-is-garbage-collection-java/