Python Server Herd

[a]CS131 UCLA, , , , ,

Abstract

This paper will explore the Python asyncio asynchronous networking library as a candidate for replacing part or all of the Wikimedia platform. We will evaluate its effectiveness as a scalable solution, and compare it to Java and Node.js. We will also consider asyncio performance and effectiveness as a framework for a proxy server herd.

1. Introduction

Wikipedia is an free online encyclopedia that hosts over 6.6 million articles in English.

Based on Debian GNU/Linux, Apache, Memcached, MariaDB, Elasticsearch, Swift, and PHP+JavaScript, the Wikimedia infrastructure uses 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.

While this works fairly well for Wikipedia, we will explore a new Wikimedia-style service where: article updates occur with high frequency, access will be required via various protocols (not just HTTP or HTTPS), and clients will tend to be more mobile. With this new service, it looks like the PHP+JavaScript application server will be a bottleneck and have poor scalability.

A potential solution is an "application server herd," where multiple application servers communicate directly to eachother as well as the core Wikimedia database and caches.

The interserver communications are designed for rapidly-evolving data (ranging from small stuff such as GPS-based locations to larger stuff such as ephemeral video data) whereas the database server will still be used for more-stable data that is less-often accessed or that requires transactional semantics. With this design, servers will communicate (directly and indirectly) by propagating interserver messages. This communication will prevent servers from consulting a central database.

For this implementation, we will explore Python's asynchronous network library, asyncio. asyncio is an event driven module that runs in a synchronized environment that focuses primarily on optimizing I/O-dominant environments. In this paper, we will explore the pros and cons of asyncio and look at a prototype application that proxies the Google Places API.

2. Python vs. Java

Python and Java, two widely used programming languages, are both considered high-level, general purpose, and object-oriented. As such, each language, to varying extents, abstracts away the explicit use of pointers, memory allocation (and freeing), to varying extents data-types, call stacks, etc., that low-level languages (e.g., C/C++) rely heavily upon. Instead, each make large use of higher abstractions like objects and garbage collection.

While Python is a run-time interpreted language, Java is a compiled language (similar to C) making it, in many cases, significantly faster than Python. Both, however are conveniently portable, each compiles to bytecode that is then run by corresponding interpreters (a Python interpreter for Python and a Java Virtual Machine (JVM) for Java).

Here we will consider three main differences between the two languages: Type checking, memory management, and Multi-threading.

2.1. Type Checking

2.1.1. Python Run Time Type Checking

As a run-time interpreted language, Python is dynamically typed. While on the surface, dynamic typing makes Python easier to write clean and concise code, there is a major drawback: maintainability. For large code bases, where types are not explicitly declared, ambiguity about (perhaps non-descriptively named) datatype can lead to error-prone code. To make matters worse, unless the code-base is heavily tested, it is possible these errors go unnoticed until run-time in a production environment. This type of error can make debugging a nightmare and ultimately lead to wasted time.

2.1.2. Java Type Checking

Java, on the other hand, is statically typed. This means Java all data-types must be known at compile time. Further, Java will not compile unless all variable assignments within the code base are consistent. While this requires extra code to be written, it also requires extra thought and prevent type mismatches.

2.2. Memory Management

As high-level languages, both Python and Java remove from programmers minds, notions of physically managing the allocation and freeing of memory for arrays, objects, etc.

This essentially means that the programmer (generally speaking) does not need to: 1) explicitly request memory from the underlying operating system; 2) keep track of how many references to allocated memory are around; 3) return memory to the operating system. By-and-large, this helps to avoid headaches like null-pointer exceptions and memory leaks.

While this convenient abstraction allows for great ease of use, there are drawbacks worth noting.

2.2.1. Python's Reference Counting Garbage Collection

Python achieves memory management with reference counting, a rather simple concept. When an object is created, Python maintains a count (starting at 1) of the number of references other objects make to it. Each time a new reference to the object is created, Python increments the objects reference count. Similarly, each time a reference is removed, Python decrements the objects reference count. Once a reference count goes to zero (0), Python reclaims the memory (garbage collects) the object was allocated. A main drawback here is self (circular) references. Unless programmers remember to break self-references, that objects reference count will never reach zero which effectively results in a memory leak.

2.2.2. Java's Generational Garbage Collection

Java, on the other hand, has a more complex Memory Management scheme with generation-based copying collection. Java runs with the notion that objects tend to be short lived and so most garbage should remain relatively local. Rather than sweeping through all memory to locate garbage, Java has only to sweep through a small subset of memory the program works with. These are called minor garbage collections. When the JVM runs out of space, it requests more space from the OS, copies live roots (a list of hot pointers to objects) and the underlying object to the head of the newly allocated memory, effectively compacting memory while simultaneously "aging" memory into generations. From time to time, the JVM also produces major garbage collections in which all generations are swept and when needed compacted.

All Java garbage collections and aging are, what Java calls, "Stop the World" events—the normal execution of the program is halted while memory is reclaimed. While these operations can consume significant CPU time, programmers can optimize program design to ensure efficient use of cache lines and pages.

2.3. Multi-threading

Another fundamental difference between Python and Java is concurrency, or the lack there of in Pythons case. While both languages support multi-process parallelism, only Java supports multi-threading parallelism (on multiple CPU cores). Multi-threading can significantly decrease wall clock time execution of programs, particularly for CPU-intensive programs.

2.3.1. Java and Multithreading

By and large, Java is designed to run fast code with convenient high-level abstractions that other systems-languages, like C, do not implement. In the nature of performance, Java, like C, offers a threading library that implement actual multi-core, parallel processing of data. That is, Java allows pure parallel computing capabilities running within a single process. This feature alone is extremely beneficial for applications running CPU intensive operations.

For high network, or more general, Input-Output (IO) applications with relatively light CPU usage, however, multi-threading will likely have similar performance to using a package like asyncio in Python, for reason to be discussed shortly. Java provides a wide host of straightforward keyword, method, class, and interface mechanisms to manage parallel program design while keeping developers focused on writing clean readable code. Such abstractions help move programmers further away from "raw" threads and transfers control to the JVM for thread execution.

2.3.2. Python and Multithreading

The second-closest Python gets to allowing multi-threading is a (misleadingly named) module called threading. The Python language implements a Global Interpreter Lock (GIL) that essentially limits one Python thread to run at a time. Since this is built into the very language itself, it is difficult to get around, though not impossible for languages like C and other implementations of Python. The threading module comes in particularly useful in high IO environments that require little CPU time. It should be noted, however, that the first-closes Python gets to multi-threading is a (accurately named) module called multiprocessing. This module allows multiple processors to run independent Python programs in parallel, however processes require significant resource overhead.

2.4. Python vs. Java Conclusion

While all around Javas overall performance outweighs Pythons, our application server herd is dominantly IO (network) bound with minor emphasis on CPU usage. Although we can build a similar Java-equivalent server herd thinking threads will make the program faster, they will not. In the context of CPU time, either version of the application herd will spend most time spinning.

Also within the context of our server herd, it is worth noting that there is a version of Python, namely Jython, that compiles down to Java bytecode and runs natively within the JVM. This can be a major pro for deploying a server-herd in Python and later down-the-line deploying a CPU intensive application in Java and requiring both platforms to operate natively.

3. asyncio vs. Node.js

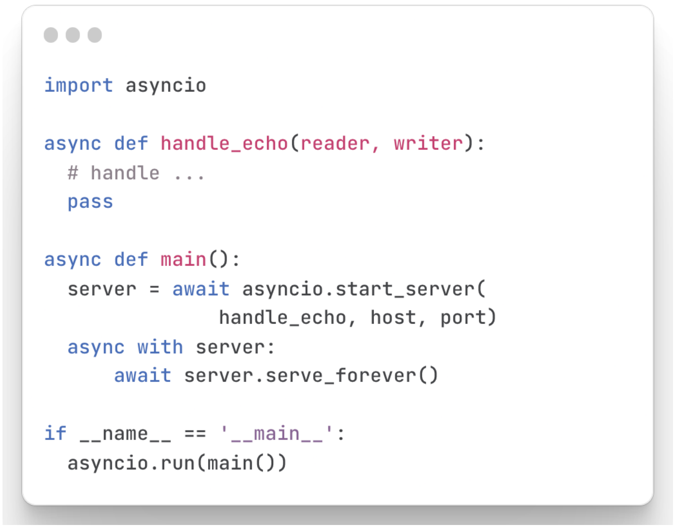
3.1. asyncio

asyncio is a Python library to write "concurrent" code using async/await syntax and is used as a foundation for Python asynchronous frameworks. As the name suggests, asyncio is fit for IO-bound (not CPU-bound), high-level structured network code.

asyncio was introduced in Python 3.4 and takes advantage of IO/network latency to schedule and cycle-through a queue of coroutines on an event-loop. Though asyncio itself is not parallel, it increases performance by managing (and polling) I/O over multiple TCP connections (via sockets).

Any Python methods intended to be a coroutine must be declared with an `async` keyword before a function definitions. All

Python methods that *call* an async method, must themselves, be an async methods. Further, ever async method call must be `awaited`. Following is an example from the server herd application:



```python
import asyncio

async def handle_echo(reader, writer):
  # handle ...
  pass

async def main():
  server = await asyncio.start_server(
                handle_echo, host, port)
  async with server:
      await server.serve_forever()

if __name__ == '__main__':
  asyncio.run(main())
```

Figure 1:

The async keyword allows methods to be suspended in order for other coroutines to be executed. Similarly, the await keyword directs the caller to suspend its execution until the called method returns. In this way, the asyncio package maximizes throughput of IO-bound programs, making this a favorable solution for the server herd.

3.2. Node.js

It is important to note, Javascript like Python, and unlike Java, is single-threaded. As such, it makes sense to compare and consider Node.js. Unlike Python, Node.js is inherently asynchronous. This means that nothing blocks (except when I/O is performed using synchronous methods of Node.js standard library). Similar to asyncio, Node.js relies on an event loop and the same general concept as outlined above apply here.

Node.js and Python asyncio are, for all intent purposes, the same. The each have async and await keywords that perform the same duties. The biggest difference is that Node was specifically designed from the ground up to perform network IO while Python was not. asyncio was added later to meet the demand of network-bound IO. Aside from this, for the purpose of the server herd, either is as good as the other.

4. asyncio as a Prototype

As shown in the code snippit above, it is relatively straightforward to define and await async methods as well as start asyncio servers. Using the `server.serve`$_{f}orever$ command allows a server to accept TCP connections from herd servers and clients alike until the server goes down. All the explicit network handling is abstracted away and left to the server. After a herd server establishes a TCP connection, it calls a handler method to handle the connection.

The server herd application performs exactly as outlined in the specs. There are three commands that enable client-server communication:

1. IAMAT [client-id] [[latitude][longitude]] [client-sent-time]
2. AT [server-id] [clock-skew] [client-id] [[latitude][longitude]] [client-sent-time]
3. WHATSAT [client-id] [radius] [number-entries]

Server-server communication relies upon `AT` commands to propagate `AT` messages from clients.

For the following discussion, we assume all commands are sent with proper formatting and are valid.

When a server receives (from a client) an `IAMAT` command, the server updates any cached records from the client, responds to the client with an `AT` message, then propagates the message to other servers in the herd it is able to communicate to.

When a server receives (from a client) a `WHATSAT` command, if the server has a record of for the client, the server will query the Google Places API and return an `AT` message followed by a single newline followed by the response in json format followed by two newlines. If the server does not have a location for the Client, it will respond with a `?` response followed by the `WHATSAT` message received.

When a server received (from a peer server) an `AT` command, if the server has already seen the message, it does nothing, otherwise the server will update it's record of the client's location and propagate the message to other servers in the herd it can communicate with.

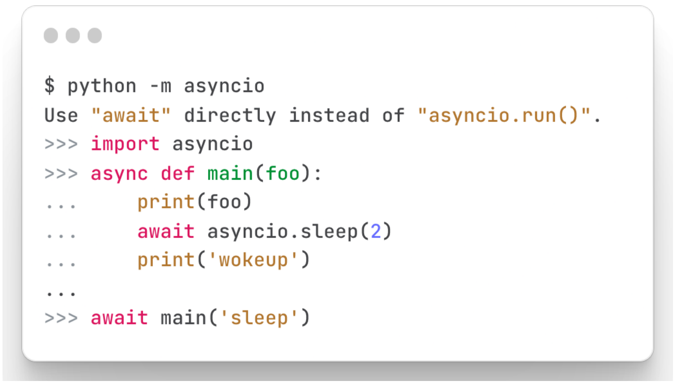
5. Performance

asyncio is a great fit for prototyping since our application is strong IO-bound. As mentioned earlier, however, should we want to perform more CPU-intensive tasks, asyncio will not help, nor will Python's threading module. Generally speaking, Python is not great for CPU-bound tasks. That said, also mentioned earlier, there is a version of Python, namely Jython, that compiles to Java bytecode and runs in the JVM, so this could be something worth looking into.

6. Reliance on Python 3.9

For Python 3.9, several changes were made to asyncio. Based directly on doc.python.org: due to significant security concerns, the *reuse_address* parameter of asyncio.loop.create_datagram_endpoint is no longer supported. A new coroutine `shutdown_default_executor` that schedules a shutdown for the default executor that waits on the ThreadPoolExecutor to finish closing. `asyncio.run()` has been updated to use the new coroutine. An asyncio.PidfdChildWatcher was added—aLinux-specific child watcher implementation that polls processess file descriptors. A new coroutine asyncio.to_thread() was added and is mainly used for running IO-bound functions in a separate thread to avoid blocking the event loop. When cancelling a task due to timeout, asyncio.wait_for() now waits until the cancellation is complete. asyncio now raises a TyperError when calling incompatible methods with an ssl.SSLSocket socket.

There is one other handy feature that was added, namely an asyncio REPL that can be run from the command line with the command `python -m asyncio`. This feature allows flexibility while learning the asyncio module without requiring a full fledged module. We can run code without haveing to invoke asyncio.run() as we did in the figure above. That is we can do as follows:



```
$ python -m asyncio
Use "await" directly instead of "asyncio.run()".
>>> import asyncio
>>> async def main(foo):
...     print(foo)
...     await asyncio.sleep(2)
...     print('wokeup')
...
>>> await main('sleep')
```

Figure 2:

7. Conclusion

Provided no CPU-bound requirements are needed from the server herd explicitely later down the road, asyncio is recommended for the proxy server herd application. asycio makes use of an event loop that allows for multiple IO-bound asynchronous events while not consuming CPU time as IO is performed.

Python, compared to Java, is a slower language overall. As a language, while it is easier to write nice code, Python's dynamic type checking, slow memory management model, and lack of pure multithreading ability on CPU-bound tasks make Python more suited for application such as we have here.

8. References

About Node.js
https://nodejs.org/en/about/

asyncio — Asynchronous I/O
https://docs.python.org/3/library/asyncio.html#module-asyncio

Welcome to Wikipedia
https://en.wikipedia.org/wiki/Main_Page

Project. Proxy herd with `asyncio`
https://web.cs.ucla.edu/classes/winter23/cs131/hw/pr.html

Java (programming language)
https://en.wikipedia.org/wiki/Java_(programming_language)

Python (programming language)
https://en.wikipedia.org/wiki/Python_(programming_language)

Alexander VanTol. Memory Management in Python
https://realpython.com/python-memory-management/#the-default-python-implementation

What is Jython?
https://www.jython.org/index

Brad Solomon. Async IO in Python: A Complete Walkthough
https://realpython.com/async-io-python/

Jim Anderson. An Intro to Threading in Python
https://realpython.com/intro-to-python-threading/