Asyncio: A Feature-Rich Module for Application Server Heards

Alexander West, *R&D Team Lead*

Abstract

An "application server herd" is a novel approach to handling rapidly-evolving client traffic involving multiple application servers communicating directly to each other as well as via the core database and caches, each for different types of data. In this test, a sample server-heard application was set up using the Python module *asyncio* to write a simple and parallelizable proxy for the Google Places API. The development process involved iteratively implementing features until single-server capability was complete. Then, the flooding feature was added allowing intercommunication between servers. This allowed for easy scalability as servers with clean caches could be added on demand. It was found that due Python being the language of choice, the development process is simpler to follow and the codebase is smaller in size making it more manageable.

1. Implementing a Server Herd

1.1. Asyncio

Asyncio is a Python-based module that "is a library to write concurrent code using the async/await syntax. It provides a set of high-level APIs to:

- run Python coroutines concurrently and have full control over their execution;
- perform network IO and IPC;
- control subprocesses;
- distribute tasks via queues;
- synchronize concurrent code;" [1]

This module is highly adaptable and provides useful abstractions when writing database software. Similar to other asynchronous code, this module uses "async" statements which return a future object. *Asyncio* implements different options for task scheduling so it handles waiting on asynchronous functions. Every asynchronous future object must be waited on using the "await" statement. *Asyncio* handles this using "coroutines" which are able to run simultaneously with other parallel code. The library also implements network I/O capabilities taking advantage of the asynchronous features. These make sending and receiving data simpler and more predictable.

1.2. Challenges

Several challenges were met while writing the sample Google Places server-heard proxy application.

Because Python is an interpreted language, debugging errors was a challenge. Since the interpreter binary (compiled from C) executes many instructions for each line of Python code, it is challenging to figure out where exactly an error is occuring. This is further complicated by the fact that many Python functions are high-level and it is hard to debug the internals of each module.

Another challenge was getting used to the asynchronous nature of *asyncio* tasks. Using the *async* and *await* keywords is not common in normal Python applications so figuring out the syntax and semantics of them required a lot of research. Additionally, the way that *asyncio* runs tasks in coroutines and how *asyncio.run()* executes certain tasks in parallel was a hard concept to get used to. It is different from other parallel code like a "thread pool" that we would see in other types of applications.

Finally, determining which functionality to add into which class was difficult. With the way Python handles classes, there is a lot of passing by value going on. So when trying to figure out which class should be storing the cache, it was a tough choice to gauge how the data should be passed around.

Working with asyncio can require a different way of thinking about program structure and control flow, meaning a significant amount of design and development effort may be needed. As a result, using asyncio effectively requires a good understanding of asynchronous programming techniques and the Python ecosystem.

1.3. Application Heards with Asyncio

One of the main benefits of asyncio is that it is very scalable in that it allows asynchronous execution of tasks separately from everything else. This can improve the responsiveness of the program as it allows a single thread to handle a large number of concurrent connections. In the context of a server, this is beneficial as each client will be served separately based on each asynchronous task. And this approach works no matter how many clients connect.

However, it's important to note that asyncio is not well-suited for CPU-bound tasks that require significant computational resources. So if the server needs to do a significant amount of processing after receiving a request from a client, then the performance may be slow. This is due to how asyncio implements its event loop. When it executes, it is within a single thread, and running CPU-bound tasks within that thread can cause the entire event loop to block, leading to poor performance and responsiveness.

In the case of the application heard, multiple instances of one Python program will be running on multiple servers and they will communicate via requests instead of a shared database or caches. This allows scalable distribution of the computational load between different servers and no duplication of work has to be done.

2. Asyncio Details

2.1. Performance

The performance improvements of *asyncio* over traditional approaches are thanks to the asynchronous nature of its event loop. With a high number of requests, it is beneficial to accept each one in real time and then asynchronously let the server finish processing the request.

A significant disadvantage to *asyncio*'s performance is the fact that it's written in Python. As is the nature of interpreted languages, it can be orders of magnitude slower than a compiled language. But, as we're dealing with long delays in terms of I/O anyways, this can be overlooked.

Another disadvantage is the fact that there is a large overhead in setting up servers for different connections. With a small number of connections, a multithreaded approach could be more beneficial. But in the case we have here where the number of requests is expected to be high, this is not an issue.

2.2. Python Version

The "async" and "await" keywords were only introduced to Python in version 3.5 [3]. This means that in order to write readable *asyncio* code, you must be running above version 3.5 which may render the approach impossible for some applications unwilling to upgrade Python versions.

Additionally, the *asyncio.run()* function was only introduced in Python version 3.9. This function is meant to schedule and be responsible for asynchronous tasks such as those used in the test application herd. So the reliance on these later Python versions is pretty important.

2.3. Compared to NodeJS

NodeJS is built on the asynchronous paradigm. So when it comes to asynchronous functions, it is the standard coding style in Node while it is an exception in Python. [4]

Both *asyncio* and Node.js are designed to handle a large number of concurrent connections efficiently. Node.js uses a single-threaded event loop to manage asynchronous operations, while *asyncio* uses a multi-threaded event loop to handle concurrent tasks. This means that *asyncio* can take advantage of multiple CPU cores and distribute tasks across them, while Node.js is limited to a single CPU core.

When the main event loop in Node.js is running, it is continuously polling for new events, such as the completion

of an I/O operation or the arrival of a new request. Upon the execution of an event, the corresponding callback function is called, and any data associated with the event is passed to the function. Since Node.js doesn't block the execution of the main thread during I/O operations, it can, like *asyncio,* handle many concurrent connections efficiently.

3. Comparing Python to Java

3.1. Memory Management

3.1.1. Python Memory

Since Python is an interpreted language, the memory management is done using the MMU implemented in the interpreter. The most common version is called CPython. This interpreter uses a garbage collector that follows a reference counting paradigm. This method of garbage collecting involves keeping track of each object on the heap and maintaining a counter of how many references to it exist. Occasionally, it will mark objects with a reference count of 0 for deletion and sweep over and free the memory.

This approach has a major drawback. A cyclical reference would mean that this algorithm will not be able to free cyclical references. The garbage collector is responsible for detecting and breaking cycles of objects that reference each other and are therefore not reachable by any other means. To address this, CPython also occasionally traverses each object in a tree starting from the root object marking all objects that are reachable, and then freeing any objects that are not marked as reachable.

In addition to reference counting and garbage collection, CPython also uses several other techniques to manage memory, such as object pooling, which reuses memory for objects of the same type that are created and destroyed frequently, and a variety of memory allocation strategies optimized for different types of objects and memory usage patterns.

3.1.2. Java Memory

In contrast to Python, Java is a compiled language. But, it also employs a garbage collector for automatic memory management. The garbage collector in Java works similarly to CPython's in that it tracks objects on the heap and frees up memory that is no longer being used. However, Java's garbage collector uses a tracing algorithm to find objects that are no longer reachable, rather than relying solely on reference counting.

The tracing algorithm used in Java is called a mark-and-sweep collector. This algorithm traverses all objects on the heap, starting from a set of root objects, marking all objects that are reachable, and then freeing any objects that are not marked as reachable. This algorithm is capable of handling cyclic references and does not require the programmer to manually break cycles.

Java also employs lots of strategies to allocate memory optimized for different types of objects and memory usage patterns. For example, it uses a technique called generational garbage collection, which separates objects into two or more generations based on their age and allocates memory accordingly. This allows the garbage collector to focus its efforts on the younger, more frequently allocated objects, while leaving older, less frequently allocated objects to be inspected by the garbage collector less frequently.

Java gives more flexibility to the programmer in including a (now deprecated) *finalize()* method. This allows custom operations on an object before it is deleted in the garbage collector. Overall, Java’s memory management makes it simpler not to focus on control flow rather than memory management.

3.2. Types

3.2.1. Python Types

The typing system Python uses is dynamic typing, which means that the type of a variable or object can be determined at runtime. Python's type system is designed to be flexible and expressive, allowing programmers to write code that reads almost a normal spoken language that is both concise and readable.

Python supports a wide range of built-in types, including integers, floating-point numbers, strings, lists, tuples, dictionaries, sets, and more. Each of these types can be used for different specific cases. They all come with a large amount of methods and properties making them easy to work with.

To supplement its builtin types, Python also supports the creation of user-defined types through the use of classes. Classes allow programmers to define custom types that encapsulate data and behavior, providing a way to model complex real-world objects in code.

Python's type system is also designed to be dynamic, which means that objects can change their type at runtime. This can be useful in situations where the type of an object is not known in advance, or where a single object needs to be used in multiple contexts with different types.

Finally, Python provides a number of tools and techniques for working with types, including type hints, which allow programmers to annotate code with information about the expected types of variables and arguments, and the isinstance() function, which can be used to check the type of an object at runtime.

3.2.2. Java Types

Java is statically-typed. This means that the types of each object and variable are determined during compilation. Java is strict in its typing which puts type safety first and allows for a wide variety of errors to be caught before the program is even run. Additionally, once the type of a variable or object is determined, there is no way to change it. Unlike other lower-level languages, there is no type casting or no way to improperly interpret data. This makes Java a very safe language in terms of types.

Like Python, Java's builtin types are extensive. These include integers, floating-point numbers, characters, booleans, arrays, and more. Like Python, it supports custom types using classes. These can be used to model real-life behavior. But a unique feature is the support for interfaces. These are like classes but put the burden of implementation on a class that inherits another. This provides a way to define common behavior across different types.

Java provides a number of tools and techniques for working with types, including type annotations, which allow programmers to annotate code with information about the expected types of variables and arguments, and the *instanceof* operator, which can be used to check the type of an object at runtime.

3.3. Multithreading

3.3.1. Python Threading

Much of the real time spent by a program is in doing I/O and one option to address this is to use multithreading. Python's multithreading capabilities are built on top of the underlying operating system's threading API, which provides a way to create and manage threads of execution. Python has a builtin module that can be used for creating and using threads called *threading*. This provides a high-level API that allows programmers to write parallel code.

One of the main advantages of multithreading in Python is that it allows programs to perform I/O-bound tasks without blocking the main thread of execution. This can improve the responsiveness of the program, as it allows the main thread to continue executing while I/O operations are performed in the background. The Python module *asyncio* that has been the focus provides an even more powerful way to asynchronously perform I/O tasks. This allows programs to scale to handle large numbers of concurrent connections.

For CPU-bound tasks, though, Python's multithreading capabilities are throttled by the GIL, or Global Interpreter Lock. This lock ensures that only one thread at a time can be executing Python bytecode. This is in contrast to normal multithreading in which different machine instructions from the same program can be executed at the same time. There have been efforts to remove the GIL but it causes compatibility issues for existing code. [2]

For a server heard, this is not an issue as instead of utilizing parallel computation, we are running a distributed server system that spreads tasks over multiple distinct Python instances.

3.3.2. Java Theading

Java has extensive multithreading support, making it easy for developers to write programs that can execute multiple threads of execution simultaneously within a single process. Like in Python, I/O-bound and CPU-bound tasks are two instances where multithreading significantly improves performance.

Java has its own builtin multithreading capabilities. The *java.lang.Thread* class provides a high-level interface for

working with threads. This eases the process of making and managing threads of execution within a Java program.

In terms of I/O-bound tasks, multithreading can improve the responsiveness of the program. It allows the main thread to continue executing while I/O operations are performed in the background. Java's *java.util.concurrent* package provides a powerful way to perform asynchronous I/O operations, allowing programs to scale to handle large numbers of concurrent connections. This can be compared to Python's *asyncio* module.

When considering CPU-bound tasks, Java allows multiple threads to execute different parts of a program simultaneously on multi-core processors. This means that unlike Python, Java does not have an equivalent to the Global Interpreter Lock, which makes it a better choice when CPU-intensive tasks are run on multiple cores. As a result, Java programs can take full advantage of the cores, making it possible to easily achieve significant performance improvements by writing parallel code that runs across multiple threads.

References

[1] Python Foundation. "Asyncio - Asynchronous I/O." Python documentation. Accessed March 16, 2023. https://docs.python.org/3/library/asyncio.html.

[2] Campbell, Steve. "Multithreading in Python with Example: Learn Gil in Python." Guru99, January 14, 2023. https://www.guru99.com/python-multithreading-gil-example.html.

[3] "Magicstack." uvloop: Blazing fast Python networking - magicstack. Accessed March 16, 2023. https://magic.io/blog/uvloop-blazing-fast-python-networking.

[4] "Python Enhancement Proposals." PEP 492 – Coroutines with async and await syntax. Accessed March 16, 2023. https://peps.python.org/pep-0492.