Suitability of Python and asyncio for Application Server Herds

Alex Hunsader – *University of California, Los Angeles*

ABSTRACT

This study serves to evaluate whether the asyncio library of Python is a suitable means of implementation in server herd applications such as Wikimedia-style news services that would have frequent updates to articles, accesses via multiple protocols, and many mobile users. To determine whether the asyncio library is a valid way of implementing this application, a sample application was created to test out the features of both Python and asyncio. This sample application followed the basic idea of the Wikimedia-style news service with severely limited features. In addition, this implementation was evaluated by comparing the features of Python to the features of Java. These features included type checking, memory management, and multithreading. Lastly, this implementation was evaluated by comparing the approach of asyncio to the approach of Node.js. Through this study, it is concluded that Python and asyncio should be used for the creation of this application.

Introduction

Many websites use the popular LAMP stack, composed of a GNU / Linux machine, an Apache server, a MySQL database, and PHP for the logic. Websites that use this stack often have multiple redundant web servers behind a load balancer. Some popular websites using this stack are Facebook, Yahoo, and Wikipedia and its related sites. While LAMP is a very popular choice for designing a large website, it is only useful in certain situations. Other web stacks are useful in different contexts. For example, the MEAN stack has risen in popularity recently for many reasons including MongoDB's ability to store unformatted data that could not easily be stored in relational databases. Due to the large number of possible web system designs, it can be difficult to determine the best structure for a particular web system.

While Wikipedia and its related sites use the LAMP stack, which works well for its purposes, it may not work well for a Wikimedia-style news service in which updates are frequent, accesses are not through solely the HTTP protocol, and many of the accesses are from mobile devices. In this type of server, the application server would be a bottleneck. To combat this bottleneck, this study will discuss the idea of creating an application server herd that uses Python and the asyncio library to communicate between the clients and servers. In this application server herd, mobile users would be able to request and send data through any server in the herd. Any data sent to a particular server would then be routed to all servers where the data would be stored in each server's cache for fast access. The goal here is that each server would rarely have to access the database in order to server information to the users.

This study will investigate the effectiveness of Python's asyncio library for the purpose of networking in server herd applications. This is investigated through a sample server herd application using Python and the asyncio library. It is also investigated through a comparison of Python and Java's features including type checking, memory management, and multithreading. Lastly, it is investigated through a comparison of asyncio's approach and the approach used by Node.js.

Design of the Sample Application

The sample application is an application server herd where clients, or mobile users, can send a name and location, along with other information to one of five servers. A client or mobile user could then query a server, the same server or a different one, to receive the location of a user along with places, such as restaurants or shops, within a certain radius of the last location of that user.

The sample application was designed by having five servers which each server connected to at least one other server and every server connected by some path to every other server. These connections were done via asyncio TCP connections. These connections allowed for each server to send location data from a client to every other server via a simple flooding algorithm, such as a modification to Breadth-First-Search. The servers would then store any client information they received, either from the client directly, or from other servers in their local caches for fast accesses. The clients could connect to the servers to send or retrieve information via asyncio TCP connections as well.

The next few sections will detail valid commands to servers and how the server will respond in the case of invalid commands.

1. Update Commands

The servers accept inputs in two forms: IAMAT and AT commands.

The IAMAT command is used for a client to send its location to the application server herd. When an IAMAT command is received by a server, the information is propagated to the other servers via AT commands using a flooding algorithm. The IAMAT command takes the form of:
IAMAT <user id> <location> <time>
The user id is the name of the user, the location is the latitude and longitude coordinates in ISO 6709 format, and the time is the time since the epoch in POSIX format. The server will then reply with an AT message. This AT message is of the form:
At <server id> <time difference> <user id> <location> <time>
The time, location, and user id are the same as before. The server id is the name of the server to which the IAMAT command was sent. The time difference is the difference between the time the client reports and the time the server has.

The AT command is used for one server to send a message to another server. This is done whenever a server receives and IAMAT command and must propagate the information to all of the servers via a flooding algorithm. The AT command has the same format as the AT response that is detailed in the previous section. It should be noted that if the AT command is sent by a client, the server will treat the command as if it were sent by a server, that is it will propagate the information.

2. Request Commands

The WHATSAT command allows a client to query the application server herd to gather information about a particular user's whereabouts, including the location of the user and places nearby, such as restaurants and stores. The WHATSAT command takes the following form:
WHATSAT <user id> <radius> <max places>.
The radius is the maximum distance away in kilometers that the returned places should be. The radius should be no more than 50 km. The max places field is the maximum number of places that should be returned. This number should be no more than 20. Upon the receiving a WHATSAT command, the server will reply an AT response of the form described in the previous subsection, detailing the location of the user. This information is taken from its local cache. In addition, a JSON will be outputted in the same form as is outputted by the Google Places Nearby Search API. It should be noted that if a user id's location is not in the server, due to that server's crash or for any other reason, the WHATSAT command will be treated as an invalid command.

3. Invalid Commands

Any command that the server receives that does not conform to the format specified by the previous three subsections will produce no internal state changes and will return a question mark followed by a space and then the original message, i.e. *? <bad command>*.

Use of asyncio in the Design of the Sample Application

asyncio is a Python package used to implement asynchronous, concurrent programs using the async / await syntax. It provides high-level APIs to implement coroutines and tasks, handle network IO, and handle the synchronization of concurrent code. In addition, it provides low-level APIs to create and manage event loops to handle networking and run subprocesses [1].

1. Coroutines, Tasks, and Futures

The basis of asyncio is using coroutines to perform asynchronous code. A coroutine is an object similar to a thread except that it cannot be run in parallel. Coroutines allow a program to schedule code while slow operations such as IO are being performed. They are usually defined for the use of asyncio using the async / await syntax. An example of a simple coroutine is shown here:

```
async def f():
      print("hello ")
      await asyncio.sleep(1)
      print("world")
```

This coroutine cannot be run simply by using the standard syntax for calling a function, i.e. f(). Instead it must be run in one of the ways dictated by the asyncio documentation. One of such ways is running the coroutine using asyncio.run(f). Another way that this coroutine can be run is by creating a task. This can be done by using the following piece of code:

```
task = asyncio.ensure_future(f())
```

The creation of this task schedules the execution of the coroutine f.

In addition, a future is a low-level object that represents the eventual result of an asynchronous operation. It is the last type of awaitable, others being coroutines and tasks. Awaitables are simply operations that can be placed after the await keyword [2].

2. Event Loops

Event loops are the low-level alternative to using the asyncio.run() function execute coroutines and subprocess. They are primarily used for lower-level code that cannot be run well with higher-level functions. This includes network IO.

Event loops provide APIs for low-level network IO such as an API to create a server that can accept TCP connections. This can be done with the following section of code:

```
server = await
asyncio.start_server(client_connected,
'127.0.0.1', port=port, loop=loop)
```

This starts a server on the local machine at the specified port. It uses the specified event loop and runs the client_connected coroutine using a StreamReader and StreamWriter. It also provides the ability for a server to open outgoing connections to other servers, using the asyncio.open_conneciton() function [3].

3. aiohttp Library

The aysncio library can only be used for TCP and SSL network requests. It cannot be used for HTTP requests that many standard APIs use. To solve this problem, the aiohttp library can be used in conjunction with asyncio. The aiohttp library provides the means to asynchronously issue HTTP requests to clients like Google Places Nearby Search.

4. Suitability of asyncio Library

The asyncio library works very well for the purposes of this Wikimedia-style news application server herd. It allows for program to have multiple coroutines so that when one coroutine is handling or waiting for IO, the other coroutines can be running on the data processing they have to do. For example, if there are two coroutines and one of them needs to wait for the HTTP response from the Google Places API call, the other coroutine can be running.

In addition, the asyncio servers provide StreamReader and StreamWriter classes that are created from the asyncio.start_server() and asyncio.open_connection() method. Instances of these classes are passed to coroutines in these methods. This allows each of these servers to allow connections from multiple hosts simultaneously, while not having the messages interfere. For example, if clients A and B are connecting to a single server, then the server will create separate coroutines with separate StreamReader and StreamWriter instances [3].

5. Problems

Overall, using asyncio and Python was relatively easy for the purposes of implementing the application server herd due to Python's simple syntax. One main problem that people face when implementing asynchronous programs with asyncio is difficulty debugging. It is difficult to debug these programs because tasks can be scheduled non-sequentially. For example, if one coroutine is scheduled before another, the second coroutine may still finish before the first. This issue of debugging often made it difficult to find errors while implementing the sample application server herd.

Comparison of Python and Java

This section will compare some of the features and approaches of Python to the features and approaches of Java. These will include type checking, memory management, and multithreading.

1. Type Checking

There are two main ways of handling type checking: static type checking and dynamic type checking. Static checking is method used by many languages including C++ and Ocaml. It requires the type to be known at compile-time either through explicit means (i.e. type decorators like int and float in C++) or through implicit means (i.e. types can be determined through the code in Ocaml). Should types ever not match, the compiler will return an error in some way. For example, if a function takes one argument as an integer, and a piece of code tries to pass in a string, the code will not compile. On the contrary, dynamic type checking does no checking at compile-time but instead does the type checking at run-time. Should the program find a typing error during its execution it will trigger and error in some way. Some common languages that use dynamic type checking are JavaScript and Ruby.

Python uses a variant of dynamic type checking commonly referred to as "duck typing." Python does not explicitly check that the type of an object is what it should be in order to perform an operation. Instead it simply tries to perform the action and then reports an error upon the failure of said action. In addition, Python does not require type declarations and allows the mixing of types in hash tables, arrays, and other objects. Overall, this allows the same amount of code to be written in far fewer lines of code. In addition, some are experimenting with the idea of adding static type checking to Python3. One such example is mypy. In mypy, functions and variables can be decorated with type [4]. This allows for static type checking on those objects. This would allow Python to have the ease of

dynamic type checking while having the reliability of static type checking.

Java uses static type checking and requires type declarations similar to those of C++. This produces reliable code that does not have typing errors but comes at the cost of bloated code. For instance, compare the code for creating a dictionary / hash table in Python to Java:

Python:

```
d = {}
```

Java:

```
HashMap<String, Integer> d = new HashMap<String, Integer>();
```

Overall, while Java may be more reliable, Python developers can be much more effective due to the simplicity of typing within the language. In addition, Python is adding static type checking functionality to improve the reliability of code.

2. Memory Management

Python and Java both make wide use of the heap to store objects in comparison to C and C++. They however do not go about allocating memory in the same way. Python allocates memory implicitly while Java requires a call to "new" to allocate memory for the most part. While they allocate memory differently, both use garbage collectors to free the allocated memory. Before the freeing of the memory of an object, both allow the object to call finalize() methods. Overall however, the garbage collection methods of Python and Java are vastly different.

Python performs garbage collection by storing the number of references of each object at a particular time. For example, when setting b to "abc," the string "abc" would have a reference count of 1. When b is set to "def," the reference count of "abc" is now 0 again. When an object reaches a reference count of 0, its memory is immediately freed and the finalize() method is called on the object. This has many benefits including very fast garbage collection and allowing the finalize() method to be called exactly at the moment the object is no longer in user. It however does produce slower pointer assignment than in other languages like Java and is not able to catch cycles in memory such as when two objects hold references to each other, but neither is used.

Oracle Java uses generation-based garbage collection combined with the mark-and-sweep algorithm. This works by storing all new objects in the newest generation, called the nursery. At certain intervals, this nursery is replaced, and the old nursery becomes an old generation. These intervals are determined by heuristics about the program. Also, at certain intervals, the nursery is garbage collected by using an algorithm to mark-and-sweep. This is done by performing DFS on each of the roots and marking all objects that can be reached from a root. All objects that are not marked are then ready to be garbage collected. The ones that are not to be garbage collected are copied to a new buffer which will turn into the nursery. All pointers to these objects are then updated. The garbage objects in the old nursery are then discarded after calling finalize(). This garbage collection is usually done in a separate thread, making it faster than Python's version of multithreading. In addition, it is able to catch and free cycles in memory unlike Python. It however it unable to perform the finalize() function of an object as the object is no longer used.

Overall, Java performs better with regard to memory management due to Java's better performance and handling of memory leaks.

3. Multithreading

Python allows multithreading but does not allow the parallel execution of threads. Java allows for both multithreading and the parallel execution of threads.

Python does not allow for the parallel execution of threads due to its Global Interpreter Lock (GIL). This lock only allows one thread to be run by the interpreter at any instance in time. This lock is used to prevent race conditions of reference counts. Should a race condition occur, memory could be freed too soon, or never be freed at all. Since Python does not support concurrency, it is unable to perform garbage collection in at the same time as running the program. Overall, the lack of concurrency supported by Python is a disadvantage of the language [5].

On the contrary, Java allows for threads to run in parallel. This allows Java developers to write much faster code. In addition, garbage collecting can often be done in a separate thread which allows for the program to run even faster.

Overall Java handles multithreading much better than Python because Java allows for threads to run in parallel on different cores.

4. Suitability of Python

Overall, Python is slower than Java because it does not allow multithreading. This lack of ability to multithread also slows down garbage collection. Python makes up for this however through its simplicity. Python is much easier to write than Java and thus developers can be more efficient.

Comparison of asyncio with Node.js

asyncio and Node.js have a very similar approach to asynchronous programming. Both use an event loop as the underlying mechanism to schedule asynchronous code [6]. In addition, both support the async / await syntax to create asynchronous functions that can be run by an event loop. They both also have elements that represent the eventual result of asynchronous code: the future in asyncio and the promise in Node.js [7].

The main difference between using Python with asyncio and Node.js is the type of project for which the technology is being used. Python and asyncio are more reliable and safer than Node.js and Javascript and thus Python with asyncio would be a better choice for programs that need security or are very large. In addition, Python provides many libraries that can be used for a variety of different purposes. Node.js and JavaScript on the other hand can be much faster, especially for computations and for creating web applications. It is faster especially for web applications because both the frontend and the backend use JavaScript allowing for easy flow of data between the frontend and the backend.

Because this Wikimedia-style news application is likely to be large and will need to be reliable, asyncio is probably a better choice.

Conclusion

Overall, asyncio and Python are a great choice for the creation of an application server herd for applications such as a Wikimedia-style news application.

asyncio and asynchronous programming in general work well for this type of application because it allows for processing to be done while waiting for IO and it allows IO from multiple sources to not interfere.

asyncio is also suitable for this application due to its use of Python. While Python is not always as fast as other languages like Java due to its lack of ability to support parallelly running threads, Python is better in other ways. For example, Python's simplistic syntax and dynamic type checking allow for developers to write much less code to accomplish the same goal.

Lastly, the approaches of asyncio and Node.js for asynchronous programming are very similar. The main difference however is that Python is a safer and more reliable language than JavaScript. This makes the asyncio and Python approach much better for large applications such as this application server herd.

References

[1] *asyncio – Asyncrhonous I/O*, https://docs.python.org/3/library/asyncio.html .
[2] *Coroutines and Tasks*, https://docs.python.org/3/library/asyncio-task.html#coroutine .
[3] *Event Loop*, https://docs.python.org/3/library/asyncio-task.html#coroutine .
[4] *mypy*, http://mypy-lang.org .
[5] *What is the Python Global Interpreter Lock (GIL)?*, https://realpython.com/python-gil.
[6] *Node Hero – Understanding Async Programming in Node.js*, https://blog.risingstack.com/node-hero-async-programming-in-node-js .
[7] *Async/await vs Coroutines vs Promises vs Callbacks*, https://blog.benestudio.co/async-await-vs-coroutines-vs-promises-eaedee4e0829 .