Suitability of Python asyncio for Application Server Herd

Abstract

Asyncio is a Python library for asynchronous I/O. It was introduced in Python 3.4 and has been maintained as a feature since. This project investigated the suitability of asyncio for the implementation of application server herds which use flooding algorithms to propagate data. To test this, a prototype was developed in which clients send their locations and query for nearby places to another client using Google's nearby search API. The research found that Python's asyncio library is suitable for an application server herd; however, other alternatives may be better for large scale use cases.

Introduction

To research the suitability of asyncio for implementing an application server herd for use in a Wikimedia-style service, a protype of 5 servers was developed where each server was connected to 2-3 other servers. Each pair of connected servers propagated information to each other and their neighbors until all servers were updated. To accomplish this, three commands were used: IAMAT, WHATSAT, and UPDATE. *IAMAT <client_id> <location> <POSIX_time_of_req>* is used to update the servers about the location of a client. This data was stored along with which server originally received the message. *WHATSAT <client_id> <radius> <max_results>* is the command used to find nearby locations to another client. Each of these commands responds with a message *AT <server_name> <latency> <client_id> <location> <POSIX_time_of_req>* with WHATSAT responding with an additional JSON string containing the response from Google's nearby search API.

Findings

The investigation found that Python's asyncio library is suitable for implementing an application server herd due to its ability to efficiently handle asynchronous I/O, ease of development through flexible type checking, efficient garbage collector, and compatibility with Python's multithreading features.

Python Asynchronous IO

The asyncio library makes asynchronous I/O extremely simple through the use of an event loop which asynchronously runs tasks from the main thread. Because tasks are run from the main thread, this means that tasks can block the main thread which limits performance by limiting multithreading capabilities [1]. However, asyncio can still be used in combination with Python's threading library which makes it suitable for a multithreaded application.

Python Ease of Development

Python is a dynamically typed language which means that variables are assigned values and the type is inferred at runtime [2]. This makes Python convenient for developers and significantly increases the rate of development by reducing time spent ensuring correct type casting as in other languages. Python offers many different type conversion functions which removes many typing difficulties associated with strongly typed languages. As a result, Python's type checking is not only sufficient, but advantageous for the development of an application server herd.

Python Garbage Collection and Memory Management

Python uses reference counting for garbage collection which means that objects are maintained in memory until no references remain [3]. This is very convenient for developers as it removes the responsibility of allocation and deallocation which ensures that memory leaks and dangling pointers are not problematic. However, this can be problematic in the case of circular references which may result in objects never being garbage collected. In spite of this issue, Python's garbage collector is more than adequate for an application server herd as it has excellent performance and circular references can be avoided by developers.

Python Multithreading

While asyncio is single-threaded, Python supports multithreading through its threading library [4]. As a result, Python is capable of utilizing multiple threads for parallel computation which improves overall performance for many workflows. Python's global interpreter lock prevents objects from being accessed by multiple threads which may reduce performance. However, Python's multithreading capabilities are still sufficient for use in the development of an application server herd.

Prototype Problems

When developing the prototype, asynchronous writes to a log file were found to be somewhat challenging due to potentially blocking the main thread. As a result, prototype logs are only available after the server has exited.

Comparison with Java

Java is another frequently used backend language that also has asynchronous I/O, a garbage collector, and multithreading capabilities. However, Java is strongly typed which means that all variables must have type declarations. This makes Java far less convenient for developers although it adds the ability to do static checking and improves performance [5]. Java's asynchronous libraries are powerful and it supports non-blocking I/O as well [6]. Python's and Java's

asynchronous I/O are both sufficient and similar for the application server herd use case. Additionally, Java also uses a garbage collector although it is different than Python's. Java's garbage collector is a generational garbage collector where objects are segregated by how recently they were created [7]. This has many advantages including rapid allocation and collection of young objects. Similarly to Python, Java's garbage collector removes the risk of memory leaks and bugs associated with pointers as it is all handled automatically. One advantage that Java has over Python is that it does not have any problems with circular references. When comparing the two languages, both garbage collectors are appropriate for implementing an application server herd although Java may be advantageous should the need for circular references arise. Lastly, Java has strong multithreading capabilities and is not limited by GIL like Python is. However, the responsibility of placing locks on objects is entirely on the developer which may result in bugs. However, Java's multithreading performance is superior. As a result, Java is likely the better choice when developing an application server herd.

Comparison with NodeJS

NodeJS is another alternative to Python for developing an application server herd. Similarly to Python, it is dynamically typed, uses an event loop to execute tasks, and shares many of the same benefits of Python. NodeJS also supports non-blocking asynchronous I/O and has an automatic garbage collector; however, it is a single threaded language which limits performance. As a result, NodeJS would likely be a poor solution for a large-scale product due to its lack of multithreading.

Recommendations and Conclusion

Python is a convenient language with sufficient capabilities for implementing an application server herd; however, it is likely not an ideal choice as Java has better performance. Should a developer choose to use Python to develop an application server herd, any version after 3.4 should suffice although some useful features were added in Python 3.9.0. These features include to_thread() which allows for running IO-bound functions in a different thread to prevent blocking [9]. Additionally, some security concerns are addressed which means that for security, it may be wise to use a more recent version of Python. The asyncio.run() feature is also extremely useful for creating new event loops and should be used as the main entry point to the program. However, other alternatives exist and its use is not mandatory. Additionally, the python -m asyncio command is another feature that was added in Python 3.8 which allows the execution of asynchronous code in the Python REPL [10]. This feature is not particularly useful for the application server herd and would not likely affect whether or not Python asyncio was selected as a development platform. In summary, Python offers many advantages through its ease of development and may be a decent selection should rapid development be prioritized over performance; however, in the case of performance being a priority, Java is likely the best choice.

References

[1] Developing with asyncio

[2] Aggressive Type Inference

[3] Python Garbage Collection

[4] Thread-based parallelism

[5] Performance Comparison Between C, Java, and Python

[6] Java non-blocking IO

[7] Java Garbage Collection Basics

[8] Event Loop Timers

[9] What's New in Python 3.9

[10] Guide to Concurrency in Python