CS131 Project Report

Chuhua Sun
UCLA

Abstract

This paper will focus on using the Python asyncio library, which enables code to run asynchronously to develop a proxy server herd program. We will utilize various features of this library and evaluate their performances in the context of a proxy server herd application. We will also make comparisons between Python and Java as programming languages and between asynchronous technologies of Python and Node.js.

1 Introduction

Wikipedia and related websites are developed on the Wikimedia server platform, which utilizes Debian GNU/Linux, the Apache webserver behind an NGINX-based TLS termination proxy, the Memcached distributed memory object cache, the MariaDB relational database, the Elasticsearch search engine, the Swift distributed object store, and core application code is written in PHP and JavaScript. They all use 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 reasonably well for Wikipedia, we are attempting to construct a new Wikimedia-style service designed for news with some specific features. First, updates to articles will happen far more often. Second, access will be required via various protocols besides just HTTP and HTTPS. Third, clients will have higher mobility. In this new design, the PHP and JavaScript application server would become a bottleneck. If we consider it from a software point of view, our new application will generate too much pain to add newer servers (e.g., for access via cell phones, where the cell phones are frequently broadcasting their GPS locations). On the other side, if we consider it from a system point of view, the response time is too high because the Wikimedia application server is a central bottleneck on the core clusters.

In this paper, we will present "application server herd," which is a different architecture where the multiple application servers communicate directly to each other. We design a rapidly-evolving data pattern for the inter-server communications to address the mobility issue. We will use Python's asyncio asynchronous networking library as a candidate for replacing part or all of the Wikimedia platform. Ideally, this approach should be fairly good since its event-driven nature should allow an update to be processed forwarded to other servers rapidly. We will then evaluate the easiness, maintainability, reliability, and flexibility of this approach. We will also analyze Python's implementation of type checking, memory management, and multi-threading and the pros and cons of using asyncio. Finally, we will compare it to Node.js.

2 Python asyncio Library

2.1 Overview

Python's asyncio library is designed for concurrent programming based on running non-parallel single threads by writing async/await syntax. It is widely used for database connection libraries, distributed task queues, and web-servers. It contains high-level APIs to run Python coroutines concurrently with full control over execution, perform network IO, control subprocesses, distribute queue tasks, and synchronize concurrent code. Moreover, it has low-level APIs to create and manage event loops, implement protocols, and bridge callback-based libraries and code.

2.2 Implementation

In asyncio, functions with async syntax would be treated as coroutines. With the help of such a design, function execution could be suspended, and coroutines could control each other since the library is implemented by using an event loop where tasks are built upon a waiting queue with the ability to break and cycle. We could utilize the event loop for scheduling or canceling coroutines and for creating client-server transports. Since coroutines can be specified, they can run concurrently with other codes and can be suspended by other coroutines with the await syntax. Moreover, future tasks can also be added to the main loop as a schedule for later execution. Additionally, streams, an API build concurrent connections with servers based on coroutine, and transports/protocols, an API utilizes callbacks for client connections based on callback, are two major ways in asyncio to support sockets and inter-processes communication.

2.3 Pros and Cons

The Python asyncio library is strong as it allows concurrency within a single thread that enables users to have more fine-grained control over task switching. It also provides much better performance for concurrent I/O bound tasks compared to Python threading. It also allows users to write concurrent code easily by using the async and await keywords. Some of the concurrency issues are well handled within the library, so users should not worry about them. Moreover, applications have the power to respond to multiple clients simultaneously with the design of coroutines and event loops.

On the other side, although asyncio's coroutine approach is similar to threads, it does not use threads offered by the operating system. A coroutine is Python, a structure to represent thread as some code partially executes, and the remaining execution of that code could be resumed later. Thus, even though Python's multithreading is concurrent, it cannot be parallel. So if the application runs into a parallel situation, it will have some issues such as data racing since many library built-in functions are not thread-safe.

2.4 Comparison with Node.js

Dynamics languages such as Python and JavaScript do not support code parallelism. Python's GIL and JavaScript's intentionally single-threaded design cause them impossible to perform threads-based parallelism. But they can both do concurrency with specific approaches. While Python utilizes the asyncio library, JavaScript has the Node.js framework. Both of the techniques are based on event-driven asynchronous models. On one side, Python is not a programming language that designed for asynchronous tasks, performing them in other suitable languages. On the other side, Node.js is a framework designed with the focus of performing asynchronous tasks. Hence Python asyncio library would have more restrictions and compatibility issues with concurrency compared to Node.js. Moreover, asyncio and Node.js both allow users to use async and await keywords to handle concurrency code with very similar designs based on the adaption of event loops. Python asyncio library has many high-level APIs which users could easily use, but they might need to implement those functions in Node.js. Node.js utilizes Promises objects to store future completion of asynchronous tasks with return values, while asyncio uses coroutines, which are awaitable objects in Python. Generally, asyncio has the infrastructure to write single-threaded concurrent code with coroutines, multiplexing I/O access over sockets, running network clients and servers. Node.js is perfect for data-intensive real-time applications that run across distributed devices due to its light-weight and efficiency. From the point of easiness, both Python and JavaScript are relatively easy in syntax, but the learning curve for asynchronous programming is another story since it is a hard idea that programmers need to master before they could use either asyncio or Node.js to perform asynchronous tasks fairly well.

2.5 Performance Implication

As we discussed in the previous sections, it is obvious that Python asyncio library would work perfectly for some tasks but not all of them. Hence its performance highly depends on the applications. If the application is mainly about I/O access over sockets, running network clients, and servers, asyncio would provide excellent performance. Hence it fits our application with outstanding performance. On the other hand, if we are trying to use asyncio with real thread-based code parallelisms such as some CPU-intensive tasks and multicore processing jobs, its performance would much worse than other mainstream programming languages that are more powerful in handling real parallelism since those languages are generally designed for that concurrency and parallelism purpose with various optimization which Python and asyncio do not provide.

2.6 Reliance on asyncio features of Python 3.9 or later

In this section, we will discuss the reliance of the asyncio library on Python 3.9. They were various updates and changes that the asyncio library made on the 3.9 version of Python. Although we might only get in touch with a part of them in this application and changes were

generally not significant, it is still notable as it generates differences to the performance and usage of the asyncio library. For instance, for asyncio.run() function that we use to start our server in the application, a coroutine that responds to the shutdown schedule for the default executor will be used. Moreover, to safely separate threads of stream I/O tasks such as those used in our application, a new coroutine called to-thread is newly included. Moreover, some of the old features were deprecated as asyncio introduced some more advanced way to optimize the library. Generally, with the introduction of Python 3.9, the asyncio library made some notable changes and improvements. It does not require users to make significant changes when using the library.

2.7 Suitability

There many powerful built-in functions in Python asyncio library for server herd application to help users to write code in a very efficient and easy manner. For instance, the start-server function that only takes in a handle callback function, an IP address, and a port number will start a TCP server. After creation, users can await serve-forever function to send and receive messages through reader and writer streams with the built-in open-connection function to perform inter-server communications. In our application, we process multiple commands and respond concurrently in such a way. When sending a message to other servers and requesting a response from the Google Places API, the server behaves like a client and uses the await keyword to perform the job.

3 Python

3.1 Overview

This section will focus on comparing some core features of programming languages such as type checking, memory management, and multithreading between Python and Java, another popular programming language that should fit our application. While they are both object-oriented languages, Python is an interpreted language, but Java is a compiled language. We will discuss those important features in detail.

3.2 Type Checking

Python is a dynamically and strongly typed language. Python interpreter does type checking only in the run time, and types of variables could be altered over their lifetime. Python could also do static type checking and duck typing with this design, while Python will always remain a dynamically typed language. Obviously, Python's code would be easier and cleaner to write since we do not define each variable's type, but this also causes readability and maintainability issues. Moreover, we would know type errors only at run time.

On the other hand, Java is a strongly but statically typed language, requiring users to pre-define the types of all variables in use and perform type checking at compile time. Therefore, Java's code might be relatively more complex than Python. Since type checking is at compile-time, we could be aware of them before the program executes.

Hence, if applications require a huge amount of codes and several programmers to maintain, Java would be slightly better than Python since Java has higher maintainability. Since our application does not require many codes, Python is a fairly good programming language for us to write more clean and easy code.

3.3 Memory Management

Python and Java both have their automatic memory management systems that are based on stack and heap. While Python has the Python memory manager, Java has Java Virtual Machine, both functioning in a similar manner. The true difference between Python and Java when considering their memory management is how they handle garbage. While some other mainstream programming languages require programmers to manually handle garbage, Python and Java's garbage handling systems automatically allocate and free memory with no user action.

Python's automatic garbage handling system is based on an approach called reference counting. Such counters would update their value to keep track of the number of references used for objects and memory pieces. Objects' memory would be freed whenever the values of their reference counters drop to zero. Although it is a concise and effective method, it is not as reliable as Java's garbage collector as it fails to handle and free circular dependencies. So the extra process to deal with the issue would damage the performance of Python. Moreover, since the system has to check those counters' values whenever the respective objects are referenced or dereferenced, it further impacts the speed of Python. Those are some of the reasons that Python is considered slower than Java.

On the other hand, Java's garbage collector utilizes the idea of tracing based on concurrent mark sweep. The method traces the memory starting at the root to locate objects that remain referenced at any point. Other objects along the way would be collected and freed all at a time, leading to a better performance than Python's approach with the cost of high space usage. Therefore, if the program is rich in memory, Java is better than Python when handling garbage. Otherwise, Python may outper-

form Java based on the consideration of space usage.

3.4 Multithreading

For multithreading, Java allows real parallelism since the Java Virtual Machine enables multiple CPU cores to perform multithreading. This feature can be easily utilized with Java built-in keywords such as synchronized, volatile, and with various related packages for similar purposes. Hence, Java was designed with the consideration of multithreading and real parallelism.

On the other hand, unlike Java, Python was not designed to perform multithreading and does not support concurrent execution on multiple CPU cores. More specifically, Python's Global Interpreter Lock prevents multithreading since it sets restrictions that only one thread could be running the program at once to prevent data racing issues. Another notable point for Python is that it is implemented based on the C libraries, whose functions were not thread safe. So this feature might lead to many other issues when performing multithreading with Python. Generally, if our application does require true parallelism, Java is a much better option than Python. Otherwise, we could utilize Python asyncio library to perform multithreading.

4 Conclusion

In conclusion, Python with its asyncio library could be considered a wise option for this application. Since our application mainly focuses on I/O streams over clients and running servers, the asyncio library fits it very well. Moreover, Python's lack of multithreading power would not cause significant issues our application does not require real parallelism. Additionally, since the size of this application is not large, the weakness of Python's type checking performance could also be ignored. Since we can build this application with Python and asyncio library in a very concise and relatively easy way, it is a valid option.

References

[1] UCLA CS131 Winter 2021 Project page: https://web.cs.ucla.edu/classes/winter21/cs131/hw/pr.html

[2] Python asyncio documentation: https://docs.python.org/3/library/asyncio.html

[3] Python Type Checking(Guide): https://realpython.com/python-type-checking/