Proxy Herd Implementation using Python 3.10.4's `asyncio`

Dylan Phe
CS 131 – Programming Language
UCLA Spring 2022

Abstract

Just like many applications, an application server can be constructed with varieties of different architectures. As our class focuses on the importance of choosing programming language, we are tasked to dig deeper into different features and modules that are offered in our attempt to build a new Wikimedia style server designed for news where multiple application server communicate directly to each other as well as via the core database caches. This paper will lay out the epitome of my evaluation on whether Python's asyncio is a perfect choice for such implementation.

1. Introduction

In the attempt to build a Wikimedia inspired service for a news application that is capable of interserver communication, we are interested in utilizing Python's asyncio asynchronous networking library for such implementations. To some extents, we do know that such task can be accomplished using asyncio, however, we want to further investigate its strengths and weaknesses in terms of performance, simplicity, reliability, and maintainability to avoid issues that may arise for a large-scale use of the application or when the application is integrated with others to become parts of a larger one. To do this, a simple prototype of parallelizable proxy herd for a Google Places API is built using Python 3.10.4's asyncio as an example to serve our investigation purposes.

This paper will describe in detail the implementation of the prototype while taking into consideration on whether asyncio is a suitable framework for the proposed application. It is important to note that this will focus solely on asyncio for Python 3.10.4 and how such implementations vary with the older version of Python. In addition, it will also briefly address how efficient or inefficient our approach is compared to a Java based approach and the overall approach of asyncio to that of Node.js.

2. Asyncio for Python 3.10.4

2.1. Background

The asyncio module was first introduced to Python as a provisional package in the version update Python 3.4 to which it was released on March 16, 2014. As the package itself was still considered to be relatively new at the time, it was expected for the module to receive notable changes that could deprecate some of its functionalities or possibly even be removed in its entirety from Python in future updates. Nevertheless, due to its significant role and importance in providing a foundation and an API for running and managing coroutines, it was later announced in the newer update version 3.6 of Python that the module itself will become a permanent part of the language standard library and will continue to evolve as time goes on.

Built upon the concept of asynchronous IO, the standard library's asyncio package is designed for programmers to write concurrent codes with Python as a programming language for their applications. It offers a new style of concurrent programming through the uses of `async` and `await` keywords that give users the look and feel of concurrency in a single threaded application. To be specific, asyncio uses a mechanism called "cooperative multitasking" to achieve such effect using a single thread in a single process. This allow the performance of I/O bound programs to be improved considerably due to lesser amounts of overhead required like those of threading approach.

2.2. A Closer Look at asyncio module

To quote Python's 3.10.4 asyncio documentation, it stated that "asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and webservers, database connection libraries and is often a perfect fit for IO-bound and high-level structed network code." This very particular statement agrees with our notion that Python's asyncio asynchronous networking library would be a good match for our problem.

The essence of Python asynchronous programming is a coroutine, which is a specialized version of Python generator function that can pause and resume its execution at many different points before the return code without losing its state. A coroutine can be defined using the `async` keyword and is given to some scheduler, or event loop to be executed immediately or later. Such steps are taken to set stage for the `await` keyword where it pauses the execution of the coroutine function to wait for the caller of the await function to return its value before resuming. In fact, this is exactly the API offered by the python's asyncio module, which allows for such creation of an event loop that automatically schedule the coroutines. If compared to a thread, a coroutine consumes less memory and requires far less overhead to create and maintain. Therefore, it allows for a faster implementation of concurrency in terms of performance.

3. Prototype: Proxy herd with asyncio

As a prototype for the investigation, I have written a simple and parallelization proxy for the Google Places API using Python's asyncio as instructed in the prompt. This server.py code can be executed as a herd of servers that can communicate bidirectionally. Specifically, for our prototype, it consists of five servers bidirectionally communicating in such patterns as shown in the below:

Servers and Ports	Adjacent Servers
Bernard (12311)	Jaquez, Johnson, Juzang
Clark (12312)	Jaquez, Juzang
Jaquez (12313)	Bernard, Clark
Johnson (12314)	Bernard, Juzang
Juzang (12315)	Bernard, Clark, Johnson

Table 1: Proxy servers and its adjacent servers for communication.

This interserver communication is designed as a model to handle rapidly evolving data, which in our case, is the location of the clients among all the servers that are running to avoid the bottleneck issue that was mainly the concern we were trying to solve for our news applications where articles and other multimedia are often updated in a fast-paced manner.

3.1. Implementation of the server protocol

For the implementation of the proxy, I decided to code everything into a single python file named *server.py*, which can be found in the gzipped tar file named project.tgz. Since each server in our herd has a specific name, an `argparse` module is used to validate and parse the argument passed by the user when the server is starting.

To start a server, within the directory containing the server source code, execute the command below:

```
python3 server.py {Server-Name}
```

If the server's name given by the user is not one of the five servers specified, the program will receive an error for invalid server name. It will also list down the name of the five valid servers that are available so that the user can start up a valid one.

On the other hand, the server itself is implemented inside a class called ServerProtocol, which can be initialized and executed using asyncio.run(args) function. When the above-mentioned function is called, the ServerProtocol class will create a new event loop to handle any requests that are sent by the clients to the assigned port. The server will continue to be served forever listening on the assigned port for new requests unless interrupted by users through a key for Keyboard Interruption.

Within the class itself are functions that were written to handle the connection between the server and its client. Since requests are sent in form of byte streams, I decided to use the coroutine start_server(args) function to create the server and establish a connection to clients. Whenever a connection is established and a request is made by any client, it will call the coroutine handle_connection(args) function where a pair of (reader, writer) objects is given read the request sent and write back the response. For security reasons, it is always advised to encode the DataStream before sending through the networks or servers.

3.2. Handling client's request IAMAT

Our server prototype allows three 3 valid commands as requests to be handled by the server asynchronously using asyncio coroutine functions. The first one being the IAMAT command that clients can use to send its location to the server. For this request, client must specify in the argument their identity, location in terms latitude and longitude in decimal degrees using ISO 6709 notation, and their time of sending the request expressed in POSIX time, which consists of seconds and nanoseconds. The request must be a valid request; therefore, a handle_req(args) function is created to validate the requests sent by the client. For such command, each argument is parsed and validated separately to make sure that the command contains all the correct and required information to generate a response.

If the server received a valid IAMAT request from a new client ID, the server would create an instance inside the dictionary of all clients connected to the server and initialized it with their coordinates (location), their time of sent, the server it was sent to as well as the time difference between the time the request was sent and received by the server. If the request is sent from an existing client, the server would just update that client's location if only the time that the request was sent is the most recent. As a response, the server will generate an AT message specify the most recent location of that client along with the time difference and the time sent by the client.

3.3. Handling client's request WHATSAT

The second command is a query command that allow client to ask for information about the nearby places surrounding the client's location through another HTTP request to the Google server using the Google Places API. For the purpose of sending an HTTP request, I used the `aiohttp` library to create a session for sending an HTTP request to the Google sever asynchronously for a response. As for a query of nearby location, the client must specify in the WHATSAT command their identity, the radius of search and an upper bound on the amount of search results. As always, the command requires validation to make sure that the correct information is provided.

On a valid WHATSAT request, the server would respond with an AT message as well as a JSON-format message of the result received from the request that was sent to the Google Place server. To output that identical JSON-format message, I used `JSON` module to parse the response received from the Google server and limited the search results as requested by the client. The response is then joined together and output as a response following the AT message.

3.4. Handling AT command

The last of the three commands is an AT command that the server can use for interserver communication. These AT commands were responses sent by the server itself during the flooding process where each server propagate the client's location to one another in a flooding manner. For such implementation, I decided to connect each server to each of its running adjacent servers while updating or instantiate the requested client's information and most importantly their locations if the location given is the most recent. This can be achieved by checking the client time stamp of which the request was sent. If it is not the most recent location, the server will not update the client's location. It will instead log the message stating that the client's location is already up to date.

Now that we have introduced how our server flood and propagate clients' locations, I also want to add on to the point mentioned earlier in the server response of the IAMAT command. Whenever client announces his or her location to the server using IAMAT command, the server will automatically flood and propagate that client's location to all the other running servers, and this complete our implementation of interserver communications between 5 running servers as specified.

3.5. Logging

For logging, I only realized after I implemented a print_log function that there exists a library in Python that can be used for logging. I found logging to be very helpful especially in debugging the application itself during the development period. As the log shows every step and all the errors each server encountered, I have made it my main debugging tool throughout the whole process.

3.6. Problems encountered

For this project, I divided the project into numerous subtasks as described above. For the most part, I found myself struggling with how to start each subtask due to my inexperience in implementing such application using Python. The project itself is time consuming for the reason that I had to read many documentations along with checking out different examples of approaches to each subtask before I decided on the most efficient approaches that I think were best for the application.

Since I only did asynchronous programming once with Node.js, I was still not quite sure how asynchronous programming work so, it took a lot of research to understand how to correctly use asyncio for the implementation of our problem mainly due to the fact that there are multiple means and APIs to accomplish the same set of tasks. It was important that I choose the most efficient one for the implementation, therefore, I decided to use the asyncio.run(args) directly to create the event loop that automatically manage and control the execution of the coroutine functions. In addition to that, I also struggled to picture how all of these pieces comes together in terms of the implementation. With lack of experience in Python asynchronous Client/Server programming, it was difficult to initially set the structure of the code before I begin the implementation.

Another issue that I found confusing was the process of flooding and propagating the client's location to from one server to another. I was not sure at first how such algorithm is implemented in the first place because the command itself was not sent by the client but the server itself. I was struggling to see how servers can communicate to one and for a while, I did not know what to use to send a request from a server to another server. Nevertheless, I was able to figure it out and successfully implemented it after I read a few posts on site StackOverflow while I was looking for solutions.

4. Analysis of the implementation

This section will examine our notion that Python's asyncio is in fact a suitable framework to implement a Wikimedia style service for news.

4.1. Python as the programming language

Before we get into the Python's asyncio module itself, I would like to first evaluate if Python is even the most suitable language for our application in the first place. The fact is known that python is a very popular language due to its simplicity and it is no difference in our case. To my experience after having written the prototype code, I think writing such application with python is relatively faster compared to the approach used by languages such like Java or C++. The rationale behind this fast performance is that asyncio module provides a style of concurrent programming on a single thread. Like mentioned, Python asynchronous IO uses coroutine functions for asynchronous programming which only run a single thread. Therefore, as a result, it will consume less memory spaces as well as required far less overhead to maintain and manage. But there're also tradeoffs in terms of performance when choosing these languages. Languages like Java and C++ would perform faster on computational bound applications whereas Python will excel in IO-bound problems. Thus, Python would be suitable for our purpose.

In terms of simplicity, Python is much simpler in terms of syntax when compared to other languages. However, such simplicity also led to larger limitation. Fortunately, Python is also known for its flexibility thank to the extensive collection of its libraries that were made available. This is because Python has such a supportive and dedicated communities of developers that continue to further extend its functionalities through new introduction of easier APIs for all kind of problems including the one that we encountered, JSON, aiohttp, logging, and most importantly, the asyncio package itself, which is the first among Java and C++ that allow concurrent programming on a single thread. All in all, regardless of the languages, programming with asyncio is still considered to be hard.

4.2. Python's asyncio package

As we have learnt earlier, Python's asyncio module is still relatively new. In fact, it was only considered to be a permanent module in the Python standard library only a few years ago. This implies that there are potential for the package to grow and evolve. Therefore, it should be kept in mind that, in terms of maintenance, it isn't guaranteed that drastic changes won't be made to the APIs of the module itself. However, it very unlikely for such changes since most of its features are no longer considered provisional.

It can also be argued that the library itself is evolving very rapidly introducing new features, which in turns will only make such implementations easier moving forward in time. Such points can be made because as of the update version 3.10.4 of Python's asyncio package, it offers both the high-level and the low-level abstractions of many underlying operations that serve as a foundation for multiple Python asynchronous frameworks with high performance network and web servers, as well as the database connection library. With only a protocol, host and port, asyncio will allow you to serve the port as a server for connection to clients. Together with other abstractions found in the immense collection of it standard libraries, Python is indeed a perfect candidate for our implementation.

5. Concerns

5.1. Type Checking

It is mentioned that Python is simpler in syntax, however, the downside of its simplicity is that it has limited its own ability for static type checking. In other words, since Python uses dynamic type checking, type checking is performed during the runtime rather than development time. Thus, it can be a problem when dealing with varying types of input data like the one we implemented for the server. It becomes a bit of concern when it comes to the reliability of the program. But as long as a good work is being done in testing and throwing exceptions for possible type errors, Python is still as reliable as the other languages. For our prototype, I would say that it can be a hassle to validate and check type those given information like the locations and times. Then again, the issue can easily be fixed with exception throwing.

5.2. Memory Management

In terms of memory management, it is mentioned that most clients will tend to be more mobile. With this scenario in mind, the memory would be considered very limited, therefore, handling large data server with a Python garbage collector would be an ideal choice. With Python memory management, garbage collection is handled by Python itself and using the reference counting collector which continuously and immediately free our memory space. For larger applications like a network server for news, it would require a lot of memory to run and executed. Since clients tends to be more mobile, it is more efficient for the application server herd to be implemented in Python as it is much more memory efficient. Also, as mentioned earlier, asyncio uses coroutine functions which also requires less memory space that is also extra advantage for choosing Python.

5.3. Multithreading

Problems concerning multithreading often arise due to it being a huge overhead load for the operating system. Now, it is proven that using Python's asyncio tend to be faster in terms of performance because when implemented in other languages like Java, it is very likely that it is only possible to program concurrency using multiple threads. Python, on the other hand, offer a package called asyncio, which we already know is known for its new concurrency programming style on a single thread.

Now if the server is running on a powerful machine with multiple cores, threading would be a way to go about our problem but again, our clients tend to be more mobile, therefore, there aren't as many processes or threads available for such purpose. Python continues to prove itself worth to be the potential condition among other languages like Java.

6. Comparison to Node.js

Just like asyncio, Node.js is an asynchronous framework for network server of the JavaScript language. It is also true that both Python's asyncio module and Node.js are only recently developed but it is noticeable that Node.js received quite the reputation for its asynchronous programming for many web-based applications. Now the same cannot be said about Python's asyncio module but when it comes to a server application like the one, we are implementing, Python would slightly be a better choice as it is better at handling the problems of our applications.

In terms of languages, Python is an object-oriented language and Node.js, which itself is the asynchronous framework for JavaScript, is also an object-oriented language, sort of. JavaScript does not natively support default object-oriented processes although it has a prototypal inheritance method, which allow programmer to modify the prototype of anything they define. Of course, it would easier to code in Python compared to Node.js, however, both has it own quality and tradeoffs as Node.js is mainly used for construction of web applications.

7. Conclusion

Python's asyncio is a suitable framework for server herd implementation. As proven by both the design of the prototype and through the evaluation of the implementation, it can be said that in terms of performance it can be faster than most when operate on mobile devices since it is a single threaded framework of asynchronous IO. However, like mentioned above, depending on different platform and the scale of the application, different languages also offer different tradeoffs that should be considered.

8. Refereneces

[1] Anderson J. (2019). *Python vs C++: Selecting the Right Tool for the Job*
https://realpython.com/python-vs-cpp/#memory-management

[2] Python 3.10.4 Documentation. (2021) *asyncio – Asynchronous IO*
https://docs.python.org/3/library/asyncio.html

[3] Python 3.10.4 Documentation. (2021) *Coroutines and tasks*
https://docs.python.org/3/library/asyncio-task.html#asyncio.run
[4] Solomon, B. (2019). *Async IO in Python: A Complete Walkthrough*
https://realpython.com/python-vs-cpp/#memory-management