CS 131 Project Report Proxy Herd with `asyncio`

Yunze Long
CS 131 Fall 2020, Discussion 1F

Abstract

In this project, we aimed to explore the implementation of an application server herd using Python's `asyncio` library, specifically in version 3.9.0.

The proxy herd implementation consists of multiple servers that communicates between each other while processing input from clients. In this report, we will mainly focus on the implementation using Python's `asyncio` library, a comparison between the implementation using Python and an alternative Java-based approach, and finally the `asyncio` library will be compared with Node.js.

1. Introduction

Wikipedia and its related sites are based on the Wikimedia server platform, which is based on GNU/Linux, Apache, MariaDB, and PHP+JavaScript, using multiple, redundant web servers behind a load-balancing virtual router and caching proxy servers for reliability and performance. While this works well for Wikipedia, if the context were to be switched to creating a Wikimedia-style service designed for news, the PHP+JavaScript application server would likely become a bottleneck, potentially slowing the performance, and consequently leading to a non-satisfactory system.

Therefore, the application server herd is proposed as an alternative approach. In this different architecture, the servers communicate amongst themselves directly, eliminating the need to communicate with database for rapidly updating data, and thus relieves the system of the possible bottleneck condition. Meanwhile, there are different approaches to implementing this architecture, this report will mainly focus on the implementation of proxy herd using Python's `asyncio` library, where Python's type checking, memory management, and multithreading will be explored, analyzed, and finally compared with Java, and the `asyncio` library will be then compared with Node.js in JavaScript.

2. Server Herd Implementation

2.1 Prototype Server Herd Design

In this report, we are asked to create a server herd that acts as a parallelizable proxy for Google Places API. The server herd consists of 5 individual servers: Hill, Jaquez, Smith, Campbell, and Singleton, which communicate amongst themselves through a TCP socket with the following restrictions:

1. Hill talks with Jaquez and Smith.
2. Singleton talks with everyone else but Hill.
3. Smith talks with Campbell.

Graphically, as shown in Diagram 1 below:

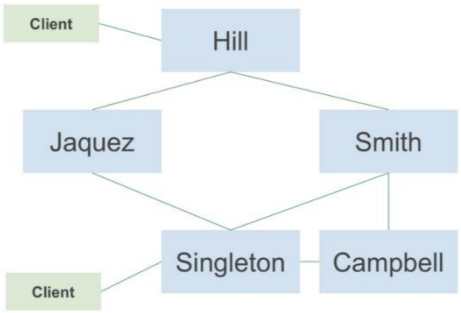

Diagram 1 (Server Communications)
(Adapted from Discussion 1B Week 9 Slides)

Clients can communicate with any server to send either an IAMAT or a WHATSAT requests, any other input that does not match the IAMAT or WHATSAT request formats would be considered invalid, and the server will respond with:

?<client input>

Where client input is the invalid request received by the server.

2.2 IAMAT Requests

A valid IAMAT Request is of the form:

IAMAT <Time Difference> <Client ID>
<Location> <Timestamp>

Where Time Difference is '+' or '-' followed by a float, Client ID is a string representing the identity of client, Location is coordinates expressed in ISO 6709 notation, and Timestamp is POSIX Time.

In the case of an IAMAT request, the server will record the information supplied by client and make corresponding updates, and then broadcast the update to the other servers to keep data in the proxy herd synchronized by using a flooding algorithm.

The broadcast message is of format:

AT <Server> <Time Difference> <Client ID>
<Location> <Timestamp>

Where Server is the server receiving the information, and other information is identical to the above case.

To prevent endless circular flooding, before each broadcast happen, the information within the broadcast will be compared with the currently existing information regarding the client, and if the message is identical, no new broadcast will be created from that server, thus terminating the broadcast cycle.

After each server is up to date with the new information, the server that received client input will then respond the client with an identical message it used to broadcast, thus confirming that updates are done within the proxy herd.

2.3 WHATSAT Requests

A valid WHATSAT Request is of the form:

WHATSAT <Client ID> <radius> <bound>

Where the Client ID is a string representing the identity of client, radius is an integer representing the search radius in kilometers and bound is an integer that sets the maximum nearby locations returned.

In the case of a WHATSAT request, the server will query Google Places API, and receive nearby locations within the given radius in JSON format. The server will then use the json library in python to extract useful data and return them to the client.

3. The `asyncio` Library

The `asyncio` library provides high-level API for concurrent programming, specifically multi-threaded programming in Python. Multithreading in python is in fact concurrent programming, and there can only be one thread being run at the same time (this will be further explained in the Multithreading section). The `asyncio` library achieves this by using an event loop, as shown in Diagram 2 below.

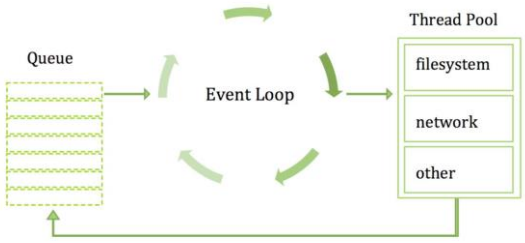

Diagram 2 (Event loop)
(Adapted from Discussion 1F Week 7 Slides)

The essence of event loop is that each thread runs a coroutine that can be called to yield the lock to other coroutines and wait, and in the meantime, the next coroutine will be picked out by the event loop and executed.

A coroutine is defined using the **async** keyword, and in this coroutine, the **await** keyword would allow the coroutine to stop executing, yield the lock, and wait.

3.1 Advantages

The `asyncio` library provides a convenient interface for setting up servers and opening ports for TCP sockets while providing an elegant way to handle muilti-threaded operations by introducing coroutines and the event loop. When multiple coroutines fight over a lock, instead of having all of them up and running, maybe spinning for the lock, which would potentially be a waste of resources as only one coroutine gets executed but the spinning also consumes CPU resources, by calling await, the coroutines can be voluntarily yield, which would conserve the wasted CPU resources and focus on the task that acquired the lock. The best part is that all these low-level operations are already written and wrapped inside a quite intuitive high-level API, and this makes implementation much less troublesome – as we did not have to write our own TCP socket or the yield features.

Aside from the easier implementation, the introduction of coroutines also help with performance. Since each connection is treated as a coroutine and would be put into the event loop, the communications between servers can be updated rapidly, which is a good fit for our purpose of designing a small but rapidly updating server herd.

3.2 Disadvantages

One of the most visible disadvantages is that the `asyncio` library does not support HTTP, so we had to use the `aiohttp` library to help implement the code for using Google Places API. Nevertheless, since the `aiohttp` library is fully compatible and makes up for this issue, this disadvantage is not severe.

Moreover, since functions are made asynchronous, they are very likely to execute out of the expected order, and this could introduce a problem of asymmetric information. For example, suppose we now have 100 servers in a herd, with many clients rapidly sending IAMAT and WHATSAT requests. There is a good chance that a particular IAMAT request is sent to the server prior to a WHATSAT request, but the WHATSAT request gets processed first. In this scenario, the response to the WHATSAT request will fail to include the information the IAMAT request would add, leading to the loss of context.

Finally, aside from the possible loss of context, there is also a performance issue. While the `asyncio` library works well with a small server herd of 5 servers, the performance is still questionable when implementing a server herd on a large scale. The higher volume of communication would still have to be processed one-by-one while the rest yield, and it is likely that there will still be a bottleneck, and the performance would then be degraded in the large scale.

3.3 Dependency on Python 3.9 or later

`asyncio.run()` is deemed stable in Python 3.8, and updated in Python 3.9. While the change from Python 3.8 to Python 3.9 is minimal, we need a Python version of 3.8 or later for this feature to be robustly supported. However, this function is essentially used to execute a coroutine and return the result while automatically managing the event loop, and an equivalent code is not hard to implement using the older versions of python, it is just a little more work and more complex coding, as shown in the Python updates page.

Moreover, running `python -m asyncio` launches a natively async REPL that allows rapid experimentation with code that has a top-level await. This is also a new feature supported in Python 3.8 and later, but for our purposes, we do not really need this feature, so it is not important to rely on Python 3.8 and later for this purpose.

Although it is not always important to rely on Python 3.9 or later versions, it is also visible that in Python 3.8 there are major optimizations to the `asyncio` library, which can make programming easier, and the code tend to be more readable. Hence, it would be still recommended to use Python 3.9 or later, as opposed to writing more complex codes in earlier versions of Python.

4. Python and Java

4.1 Type Checking

Type checking take place in run-time in Python due to its dynamically typed feature. This enables the programmer to have a higher level of freedom, and consequently make programs easier to write. A notable feature is that Python allows different types to be written into the same tuple, which is a feature many statically typed languages do not provide. However, this would also mean that it is more likely for Python to encounter errors in runtime, since many errors are not checked at compile time.

In contrast, type checking in Java take place at compile time; the compiler would throw an error if inconsistent types are seen. In result, this would make programming stricter, and therefore more difficult to write for programmers, and the type castings typically are more complex. This could also mean that programmers may need to declare more variables just so they can have different types of variables. However, despite the more difficult programming,

once compiled, Java would be less likely to throw errors as all types are checked during compilation – and this would make Java programs safer in effect, as the stricter type checking tend to rule out unexpected or forgotten logic errors too.

4.2 Memory Management

Python uses a garbage collector based on reference counts, the number of variables referencing an object. Once an object has 0 reference counts, its memory will be automatically freed. Since this criterion is very simple, the garbage collection in Python is quick and unlikely to degrade performance, in most cases there is no need for a manual override. However, this feature also comes with a drawback, that it is not very efficient for circular references. As in the case of circular references, there is a good chance that the reference count never goes to 0 even if the objects are no longer in use, but Python garbage collector would not free up the memory because the reference count is non-zero, thus introducing memory leak.

On the other hand, while Java also uses garbage collector, the approach is entirely different. In Java, a "mark-and-sweep" algorithm is employed, such that each unreferenced object will be marked, and the heap is divided into 3 generations: young, old, and permanent. Newly created objects start in the young generation, many of which have short lifespans, so garbage collection from the young generation is a minor event. Then, if the object lives long enough, they will be eventually moved to the old generation, from where garbage collection would be a major event. The permanent generation would contain metadata like classes, and garbage collections from this generation would typically rarely happens. Then, the garbage collection would proceed by continuing the mark and sweep algorithm, it goes through all objects and free up the memories of the marked objects. This mark-and-sweep would happen when the application is running, but since all objects are gone through, this garbage collector is more likely to be expensive, and to some extent it may result in a worse performance.

4.3 Multithreading

As mentioned briefly in the `asyncio` section, multithreading can be achieved in python, but instead of the parallel programming we see in Java, python handles multithreading in a concurrent fashion, meaning only one thread can be executed at the same time. This is because of the use of Global Interpreter Lock (GIL). The GIL can only be acquired by one thread at the same time, and all threads must wait for their turn to use the interpreter while another thread holds the GIL.

This feature is designed due to the garbage collection in Python. As mentioned above in the Memory Management section, Python garbage collection is based on referenced count. If multithreading were to be implemented in parallel, there is risk for memory leak and garbage values, so to avoid this, multithreading must be enforced as concurrent but not parallel.

Therefore, multithreading is guaranteed to be safe and race-free in python, but the performance would also be guaranteed to be worse than other languages that allows multi-threading in parallel.

In comparison, multithreading in Java has no such limit, and threads can be run in parallel, which means multiple threads can run at the same time. This would effectively make performance better, especially in workload-heavy threads.

5. `asyncio` and Node.js

Node.js is a popular approach for building scalable web applications. Like the `asyncio` library in Python, Node.js also uses an event loop to achieve concurrent programming. However, as JavaScript is intrinsically asynchronous by nature, Node.js presents event loop as a runtime construct, so there is no function call to start the event loop, and Node.js would simply enter the event loop after executing the input script, and it will exit the event loop when there are no callbacks to perform. Therefore, Node.js is more likely to have better run-time performance than the `asyncio` library in Python.

However, although a similar approach is used for concurrent programming, Node.js is still quite

different. In python, the threads enter the event loop, but Node.js is designed without threads – this means that to achieve concurrent programming, instead of threads, we must use processes – which then requires a multi-core environment. So in essence, Node.js will have a higher requirement for the server CPU, and it will likely consume more CPU resources than the `asyncio` library in Python, as each process must have their own memory space, while the `asyncio` library allows multithreading and the threads would share one memory space and do not necessarily require a multi-core environment.

It is also worth mentioning that Node.js supports HTTP (in fact, the official About Node.js Page described HTTP as a first-class citizen), which for our prototype would be a very convenient feature. In comparison, since the `asyncio` library does not have HTTP support, an additional library `aiohttp` has to be called to the rescue. While Node.js does not require the loading and linking of another library, it is more likely to have a better overall performance than Python, in which we are likely to import multiple libraries and therefore result in additional loading/linking time.

Despite the potentially worse performance, Python's libraries also make Python more robust, so that they are more versatile when it comes to task-processing, which is an advantage when compared to Node.js.

Overall, in the prototype of 5 servers, the difference between the `asyncio` library and Node.js is minimal, but due to the better run-time performance and HTTP support, Node.js is more likely to perform better at a large scale. It is therefore more scalable than the `asyncio` library, but this comes at the cost of more CPU resources being occupied. Thus, in processor-intensive scenarios, the `asyncio` library seems more appropriate Node.js.

6. Conclusion

Throughout this report, we explored various aspects of Python as a programming language and the use of the `asyncio` library to create a server herd. Overall, for a small-scaled server herd, Python's `asyncio` library is entirely appropriate and is likely to work well due to its easily implemented concurrent programming. However, the scalability of `asyncio` library remains in question, and on a first look, Node.js seems to be able to provide a better solution in a large scale at the cost of occupying more CPU resources. Therefore, at this point, without further exploring the features of Node.js, a definitive conclusion of which approach is obviously better than the other cannot be drawn, but Python's easy implementation and reasonable performance makes it a completely suitable choice for our purpose and a strong contender especially in the implementation of a smaller scale server herd.

7. Reference

1. Eggert, Paul. Project. Proxy herd with `asyncio` https://web.cs.ucla.edu/classes/fall20/cs131/hw/pr.html
2. Discussion 1B Week 9 Slides
3. Discussion 1F Week 7 Slides
4. Aiohttp maintainers. Async HTTP client/server for `asyncio` and Python https://docs.aiohttp.org/en/stable/
5. Langa, Łukasz. What's New in Python 3.9 https://docs.python.org/3/whatsnew/3.9.html#asyncio
6. Python `asyncio` library https://docs.python.org/3/library/asyncio.html
7. What's New In Python 3.8 https://docs.python.org/3/whatsnew/3.8.html
8. Solomon, Brad. Async IO in Python: A Complete Walkthrough https://realpython.com/async-io-python/
9. Debrie, Alex. Python Garbage Collection: What It Is and How It Works https://stackify.com/python-garbage-collection/
10. Altvater, Alexandra. What is Java Garbage Collection? How It Works, Best Practices, Tutorials, and More https://stackify.com/what-is-java-garbage-collection/
11. About Node.js® https://nodejs.org/en/about/