Server Proxy Herd with Python's Asyncio

University of California - Los Angeles

Abstract

In this project, it was tasked that we use Python's asyncio library in order to test the difference in whether or not it will perform better than the classic LAMP architecture. The LAMP architecture prioritized the central server herd, in which applications would have to communicate with a single server, which often caused many bottlenecks and latency issues as a result thereof. With my design, I implemented a server model using the asyncio library and a flooding scheme, where there is no central database, but rather servers flooded information to one another. The result is comparable to that of NodeJS and benefits and concerns are both noted in this discussion.

1 Introduction

While LAMP is often times suitable for many dynamic applications that require a centralized hub of information, LAMP will easily bottleneck should servicing become too great of a requirement. However, an application server herd design was proposed as a solution to this problem. After looking into it, the solution seemed quite feasible and a good choice in order to provide a solution to the issue of the bottleneck. Instead of providing communication to a central server, the server herd architecture allows for data to be "flooded" to the other servers.

In addition to this, the Python library is a plausible method to implement and develop said servers. Theoretically, Python 3.8.2 contains and maintains a stable development of the servers, but it has not been tested in practicality. As of currently, questions of feasibility, performance, and reliability remain in the air to be further explored and discussed. Thus, the test implementation that I have created will give us valuable information on these topics. Further research will shed some light on the inner workings of the programs.

2 Asyncio

Asyncio is a python development library that allows developers to write concurrent code using the async/await syntax. It is used mainly for asynchronous frameworks where network connections, database queries, and distributed tasks are required. The main point of using this library is to further explore the usage implementing asynchronous IO using coroutines on an event loop to schedule these coroutines.

Coroutines are subroutines in a whole, but they are a specialized form of subroutines. With subroutines, one can only enter at a designated point and exit at a different designated point. However, with coroutines, one can enter, exit, and resume at many different points. This allows for callers to send data in the middle of a program. In the example code below, we see how this can work.

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)  # Any IO-intensive task here
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())  # Add to an event loop
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```
$ python3 countasync.py
One
One
One
Two
Two
Two
countasync.py executed in 1.01 seconds.
```

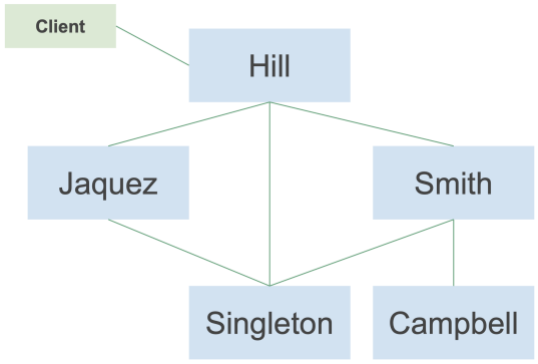
In the example above, we see that await asyncio.sleep(1) will print “One” three times before printing the second value “Two”. This is an example of how the coroutine function works in asyncio: it will not run the following lines until it is scheduled or called upon. In addition to this, we see that the coroutines are able to function almost simultaneously with a timer, which is a show of how it can jump between coroutines consistently. This is all controlled by event loops.

The event loop runs the tasks which are waiting to be completed. These tasks run a coroutine until it is completed and then proceed on to the next coroutine. This can be viewed in the source code for the asyncio event loop. As can be seen in the source code below, the event loop will run until a process finishes, thus allowing a process to decide when to yield. [1][3]

```
596         if (self._local._loop is None and
597             not self._local._set_called and
598             isinstance(threading.current_thread(), threading._MainThread)):
599             self.set_event_loop(self.new_event_loop())
600         if self._local._loop is None:
601             raise RuntimeError('There is no current event loop in thread %r.'
602                                % threading.current_thread().name)
603         return self._local._loop
```

3 Server Design

From the specifications, we were constricted to making a server herd with a specified connection design as the one shown below.



In the server design above, only the servers Hill, Jaquez, Smith, Singleton, and Campbell were required. The client will be able to connect to any server and should be able to request the data as it is called upon with the messages IAMAT and WHATSAT. The arguments were set in format of {<command> <host> <lat><long> <date-time>}.

Upon receiving these arguments, the host server should return the message as specified, where whitespace does not matter. Now, from viewing, not all servers are connected in the traditional centralized database structure, so a flooding algorithm was required in order to share data across the many servers. This flooding algorithm was an algorithm where servers would send the data they received to all networked servers they were connected to and the following servers would broadcast on and so forth until the data was shared across all servers. This brought up the issue of a loop sequence of sharing where servers could be receiving double information, but was easily handled by creating a controlled flooding scheme. This was designed so that the servers would attach their own address onto the packages ensuring that the package was not later sent back. [2]

In order to do this, much of the tutorials from discussion were used in order to set up the server in the first place. At the very basic level, the servers were started using asyncio.start_server() and looped continually using loop.run_until_complete(). The loop ran continually until a keyboard interrupt was detected, at which point the server was shut down and stopped, but without effecting any other servers in the network. The following instructions were parsed in parse.py, with the server configuration being run in config.py, and server.py handled receiving messages and passing them through the other two python files.

4 Analysis

Through this process, I learned a lot about the connections used in asyncio. All in all, asyncio is fundamental to the process in which this architecture works, however, I do believe NodeJS is a viable substitution if necessary. As a programmer that has worked with both, I can say that Python would be much easier to develop in, but further along may provide issues with troubleshooting as discussed below. However, the Python library provides near identical functionality as NodeJS and comes at the advantage that we can optimize this for our personal servers.

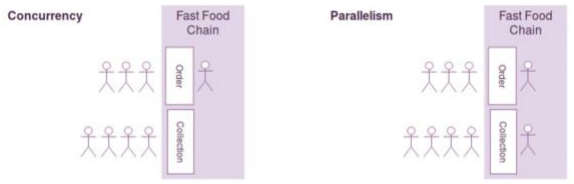
4.1 Benefits

Asyncio as we have seen in our various tests have proven quite successful with very few issues arising other than in the programming and server design. In all, I have noticed that the only issues I ran into with this design was due to misunderstanding code or forgetting to put some parameter in correctly. Once I had a grasp on how it all worked, the rest of the project was easy in that it was straightforward to implement and the server connections were easy to implement after one server was up. This allowed me to design and create routing connections quickly and gave a level of abstraction not available in NodeJS. In essence, for the programmer willing to read and put in the time for the documentation and experimenting, this is favorable, as the process will go much more smoothly since there is no need to understand as many components.

4.2 Drawbacks

With the development in Python, there are three fundamental issues to be addressed: memory management, multithreading, and type checking. First and foremost, memory management is a concern because of how Python used to handle garbage collection. Python used to have reference counts that would remove objects from the heap once it was finished being used, which created high overhead and latency issues. In recent years, Python has addressed this issue by adding Java style garbage collection only when memory is low, thus creating higher performance.

The next issue to be addressed is the capability of this specific library in concurrency. As it is asyncio, thhe documentation states that it does not support multithreading, which means it runs on a singular thread. From this, we draw up the diagram below to show a better picture of its benefits and restraints.



The benefit above is that we can still provide concurrency using asyncio, but this can create longer queues with result in bottlenecking if there is a large amount of large orders needed. However, this also means that programmers will not need to worry about race conditions or packet collisions, which could be even more detrimental to the health of our servers. This is a tradeoff, but a necessary one, as packet collision could prove to be more fatal than a slow service. [1]

Lastly, our issue is with Python type inference. In standard C and Java, we must declare our variables and their types, whereas with this implementation, we see that it is unnecessary and this proves to be as convenient as it is detrimental. This forces our Python interpreter to do work on type checking, which can result in many errors regarding types. With no types available to be seen or declared, it will prove difficult to troubleshoot, with no other warning to the programmer other than to be careful. In this project alone, I spent most of my time fixing issues with type incompatibility. [4][5]

5 Conclusion

In all, I believe this implementation should be used. It is a good implementation with near similar capabilities to that of NodeJS. After having used this, I can say that due to the familiarity of Python, it may be an even easier way to onboard new programmers looking to expand and develop further off this server, as most will know Python. The builds from all of the libraries seem fairly straightforward and reliable, with most of my tests running through capably. Ultimately, this server is reliable and will do the job very well and considering we will be running this off of our own hardware, it is highly suggestive Python is used since it has further capabilities to optimize with whatever hardware we end up using.

Works Cited

[1] asyncio - Asynchronous I/O ¶ . (n.d.). Retrieved from https://docs.python.org/3/library/asyncio.html

[2] Eggert, Paul. “Project. Proxy Herd with Asyncio.” Project. Proxy Herd with Asyncio, web.cs.ucla.edu/classes/winter20/cs131/hw/pr.html.

[3] Glossary ¶ . (n.d.). Retrieved from https://docs.python.org/3/glossary.html#term-coroutine

[4] Type inference and type annotations ¶ . (n.d.). Retrieved from https://mypy.readthedocs.io/en/stable/type_inference_and_annotations.html

[5] What's New In Python 3.8 ¶ . (n.d.). Retrieved from https://docs.python.org/3/whatsnew/3.8.html#asyncio