Server Herd with Asyncio

Tanzeela N. Khan

University of California, Los Angeles
tanzeelakhan@g.ucla.edu

Abstract

The following paper reviews my implementation of proxy herd with `asyncio` , and the costs and benefits of `asyncio` relative to other programming languages. This project involved using python's asyncio to create multiple servers that speak to each other and communicate with outside clients. I used `asyncio` to implement a to propagate messages between servers. Clients send messages to servers, and some messages will trigger the server to make a get request using python's aiohttp. While doing this project,

1 Introduction

Asyncio is a python module that allows programmers to write single-threaded concurrent code using coroutines. In our project, we implement This module provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives.

Asynchronous programming is more complex than classical sequential programming: see the Develop with `asyncio` page which lists common traps and explains how to avoid them. Enable the debug mode during development to detect common issues. [1]

2 Asyncio

My implementation of the server-herd with asyncio began with the EchoClientServerProtocol example code on asyncio's docs. This code allowed for a server and client to speak to each to each other.

In order to make get requests in asyncio, I followed this prototype from the asyncio docs:

```
import aiohttpimport asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

By default, python's functions are not asynchronous. To call asynchronous functions such as the above fetch() within non-async functions, we need to create a future. To schedule an async co-routine, we must schedule its execution using `ensure_future` function.

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

2.1 Pros

The asynchronous model allows you to initiate tasks that would normally be blocked so that we can work on other tasks without needing multiple threads. Messages arrive and depart via the network at unpredictable times - asyncio lets you deal with such interactions simultaneously.

Python has good package and library support. There were multiple options for making an HTTP get request. We also had several options for creating non-blocking open connections. [2]

Easy to see when you await something, in contrast to other libraries that sweep the magic behind the rug. As a result, using asyncio is less error-prone.

It is fairly easy to write asyncio-biased programs that run and exploit server herds. With minimal understanding, I was able to copy and paste code from the documentation and develop one in less than a week. Asyncio is fairly platonic too, unlike other non-blocking http client libraries such as twisted.

2.2 Cons

Writing code that supports both async and sync is a lost cause. Asyncio is not very fast.

"latency sensitive services which serve small requests and do not need I/O e.g serve from memory and just do computation on top do not need async. another disadvantage is complexity. even throughput is hurt by async sometimes but it makes up by the safety net it provides against threads blocking/starvation."

3 Problems

While implementing the server herding, I ran into and tackled several issues which I will outline.

3.1 Coroutine Issues

To make get request, I need to use asyncio's async fetch function. Since I was trying to call this within a non-async function defined by the Protocol class, I was running into the following error `coroutine was never awaited`. This was happening because the coroutine object being returned needed to be scheduled for execution.

I could either do this by awaiting or yielding the query function in another asynchronous function or scheduling its execution using `ensure_future` or adding it to my loop. I chose to schedule its execution in `ensure_future`.

3.2 Infinite Flooding

While implementing the flooding from server to servers, I ran into an issue where there was infinite flooding. I implemented flooding by having a for loop iterate through each of the friends of the server and opening a connection, sending a message, and closing a connection. The message would be recognized by the receiving server as a new connection and the server would then check the timestamp with its own array of clients and decide whether or not to propagate the message. The error was here.

The issue was with that the clients dictionary was an object of the `EchoServerClientProtocol` class. Each time a new connection was made to a server, there is a completely new instance of this class. Hence, a new and empty client dictionary was always created. As a result, there would never be a matching timestamp and the stopping condition is never accepted. To deal with this issue, I made the client dictionary a global variable of the server class, so that it is not recreated and persistent.

4 Python vs. Java

Python and Java are imperative languages that have great package support, making them flexible. Despite these similarities, they vary notably in type checking, memory management, and multi threading.

4.1 Type Checking

Python is dynamically typed so it has many design restrictions. Python has duck type checking. You simply declare the expression of the variable, and the variable acts that way. Dynamic type checking is easier for the programmer because the code is easily mutable. This can be an issue because it reduces the risk of undetected errors from eventually hurting your program.

On the other hand, Java has strict static checking. The user must define the type of variable when declaring it. [3]

4.2 Memory Management

Python uses a private heap for memory management. The interpreter and memory manager perform the heap management and the user has no control over it. Python uses reference counting to keep track of objects.

Java allocates memory to new objects. Java has a grabage collector that frees space when there are no longer references to objects[4]

4.3 Multi threading

Python does not have good multi-threading support. Python uses Global Interpretor Lock (GIL) that supervises resource sharing of threads. As a result, only one python instruction can run, decreasing performance time.

Java is a lot more flexible with multithreaded programs. It uses multiple cores and hyper threading. The language even has several locking synchronizing mechanisms, adding to flexibility and performance. Hence, Java has the advantage over python in multi-threading.

5 Asyncio vs. Node.js

Asyncio and Node both provide servers. Node.JS is has a single threaded event callback mecahnism. This feature makes it more efficient thn traditional threading, and programmers cna connect scripting languages such as Javascript with network programming [5]

By default, functions are asynchronous in Javascript's Node. In Asyncio, you must declare that the function is asynchronous and add it to the event loop manually, in one of the methods listed earlier. Python is typically used for projects involving data analysis, Web APIS, machine learning, and heavy data processing. Node.js is used more for general backend development.

Both are "sticky". You can put a python program inside a Node program and vice versa, but Node is more flexible in this case.

More startups and companies are shifting to Node.js over Python's asyncio. Some of these companies have APIs and products that support only some of the functionality for Python for moreso for Node.js.

6 Recommendations and Conclusion

When compared to Java, we see that Python is lacking in performance due to its version of multi-threading and dynamic type checking. For projects that require heavy multi-threading, I would recommend Java because of its flexibility. Since this project is a lot simpler and does not require multi-threading, Python is a good choice too in this case.

We saw that Python servers are typically used for Data analysis and processing. Otherwise, Node.js is used for backend development and other general use cases because of its efficiency and multi-threading capabiliites. Since this assignment involves no heavy data processing, simply calling an API and flooding a mostly response to other servers, I would recommend Node.JS over python in this case.

Overall, I would recommend Node.JS to implement this server herding project.

7 References

1. Mike D., Python 3 An Intro to asyncio, Mouse vs Python, 2016, https://www.blog.pythonlibrary.org/2016/07/26/python-3-an-intro-to-asyncio/

2. Jeff Knupp, Python's Hardest Problem, JeffKnupp, 2012, https://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/

3. Tom Radcliffe, PYTHON VS. JAVA: DUCK TYPING, PARSING ON WHITESPACE AND OTHER COOL DIFFERENCES, ActiveState, 2016, https://www.activestate.com/blog/2016/01/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences

4. DataFlair Team, Python vs Java Comparison between Java and Python, Data Flair, 2018 https://data-flair.training/blogs/python-vs-java/

5. Uroosa Sehar, Node.JS VS Python: Which is the Best Option for Your Startup, Viztech Solutions, 2016, t https://vizteck.com/blog/node-js-vs-python-best-option-startup/