CS 131: Proxy herd with *asyncio*

Fulati Aizihaer
University of California, Los Angeles

Abstract

This report describes the development of an application server herd modeled after the Wikimedia platform's architecture, implemented using Python's *asyncio* asynchronous library. It evaluates the advantages and limitations of using *asyncio* and compares it with Java's multithreading model and Node.js to evaluate its overall suitability for this type of system.

1 Introduction

This project focuses on building an application server herd that works together to manage location data and respond to user queries. The system consists of five interconnected servers that communicate with one another, handle client messages, and fetch nearby place information using the Google Places API. Instead of relying on a single server, the data is shared across the five servers using a simple flooding algorithm to allow updates to spread quickly and reliably.

The servers are built using Python's *asyncio* library, which supports asynchronous programming through coroutines and event loops. This lets each server handle multiple client connections and API calls at the same time without blocking any operations. Commands like `IAMAT` and `WHATSAT` are processed efficiently, and updates are shared between servers in real time.

The goal of this project is to evaluate how well Python's *asyncio* handles I/O-bound tasks and to compare its performance with Java's multithreading model and Node.js's asynchronous approach. This report outlines my implementation, the challenges I encountered, what worked well, and whether *asyncio* is a suitable choice for this type of application.

2 Implementation

The system is made up of five servers: Bailey, Bona, Campbell, Clark, and Jaquezl. Each server listens on a specific port and has a list of neighbor servers it can communicate with. Each server talks to only a few of the other servers instead of all of them directly.

The server processes three types of messages:

- `IAMAT`: Sent by a client to report its client ID, latitude and longitude coordinates, and timestamp. The server calculates the time difference between when the message was received and the client's timestamp, stores the client's latest location, and sends the update to its neighboring servers using an `AT` message.
- `WHATSAT`: Sent by a client to query for information about places near other client's locations based on a client ID. The server finds the client's latest location, sends a request to the Google Places API, filters the results by radius and upper bound, then sends back the response.
- `AT`: Sent between servers to share the latest client location data. If the data is new or more recent, the receiving server updates its records and continues flooding the message to its own neighbors.

All input and output operations are handled asynchronously using Python's *asyncio* and *aiohttp* libraries, so servers can handle multiple client connections and API calls at the same time without blocking. Each server also keeps a local log file to record events, incoming messages, and outgoing updates for debugging.

3 Asyncio: Pros and Cons

Python's *asyncio* library provides an efficient way to write programs that can handle multiple things at once without blocking, through the use of `async` and `await`. This makes the code easier to read and avoids the complexity of working with threads. For I/O tasks like handling messages between clients or making API calls, *asyncio* works very well.

During this project, I followed the official Python documentation and the example code to set everything up. I created a server class that could be used for all five servers with different names and ports. To connect neighboring servers and forward update, I used `asyncio.open_connection` inside the flooding function. To handle the communications I used `reader.readline()` and `writer.write()`, and used `asyncio.run()` to start the event loop. While there was a bit of a learning curve in understanding how things worked, *asyncio* was a suitable tool for building this type of system.

However, *asyncio* also has its limitations. Python's Global Interpreter Lock (GIL) means that only one thread can run Python code at a time. Because of this, *asyncio* can only run tasks concurrently but not in parallel across multiple CPU cores. This works fine for I/O bound tasks, but for CPU heavy tasks, it becomes less efficient. Another limitation is that *asyncio* only works with protocols like TCP and SSL, and not HTTP. So for this project, I had to use a separate library called *aiohttp* to make HTTP requests to the Google Places API.

4 Python 3.9+ Features

Features like `asyncio.run` were introduced in Python 3.7, and the `python3 -m asyncio` command was added in Python 3.8. These tools make it much easier to start and test asynchronous programs, as developers no longer need to manually set up and manage event loops. While Python 3.7 fully supports the core *asyncio* features needed for this project, using Python 3.9 or later does offer additional improvements to task management and cleaner code. However, it is not a must to rely on Python 3.9 or later, since we can still effectively utilize *asyncio* with Python 3.7 or 3.8.

5 Python vs Java

One of the main differences between Python and Java is how they handle type checking. Python is dynamically typed, so we don't have to declare the type of variables, and type checking happens at runtime. This speeds up the development, however it can lead to more bugs in larger programs because the errors show up only when the code runs. Java, on the other hand, is statically typed, which means you must declare types, and the compiler checks them before running the code. This helps catch mistakes early and makes Java more reliable for large projects.

Another difference is in memory management, where although they both use garbage collection, they do it in different ways. Python mainly uses reference counting to keep track of the objects, and it cleans them up when the reference count drops to zero. This works well most of the time but can have issues with objects that refer to each other, since their counts never reach zero. Java, on the other hand, uses a mark and sweep approach, which goes through the memory to find all the objects that are still being used and removes the rest. This approach is more efficient and reliable in larger applications that use a lot of memory.

Third difference is how they handle multithreading. Java allows true multithreading, where it can run multiple threads in parallel on different CPU cores, which is great for CPU heavy tasks. Python on the other hand is limited by the Global Interpreter Lock (GIL), which prevents multiple threads from running python code at the same time. That can be a problem for CPU heavy tasks, but for I/O-based systems like this project, it is not a big issue. Python's *asyncio* lets you manage many tasks at the same time without needing to deal with the complexity of real multithreading.

6 Asyncio vs Node.js

Both Python's *asyncio* and Node.js let you run multiple tasks at once using an event loop to manage I/O tasks efficiently without blocking the whole program. Python uses *async* and *await* with coroutines, whereas Node.js uses callbacks and Promises. Both are great for handling lots of I/O operations quickly. Node.js is often considered faster because it uses Google's V8 engine.

One key difference between the two is how they handle the event loop. In Node.js, the event loop starts automatically and keeps running as long as something still needs to run. In Python, you have to manually start the event loop using *asyncio.run()*. Additionally, in Node.js, an *async* function starts running right away up to the first *await*, while in Python nothing runs until you explicitly run it. Because of this, Python gives more control, while Node.js is more automatic.

Another difference is in built-in tools and features provided for each. Node.js was designed for asynchronous programming, so it comes with native support for HTTP and lots of helpful libraries. Python's *asyncio* came later, so many of its standard libraries are still blocking, and you often need additional libraries like *aiohttp* to fully use async features.

Overall, Node.js may be the better choice for more complex, I/O-heavy applications because of its built-in asynchronous support, performance, and a more mature ecosystem.

7 Challenges

Some challenges I encountered during this project involved both learning how *asyncio* worked and setting it up to fit the specific needs of my application. While I followed the documentation which provided useful example code, I still had to make several adjustments to get it working properly with the flooding mechanism and handling the *API* calls. One of the main challenges I faced was that the *asyncio* does not support HTTP requests, so I had to use the *aiohttp* library in order to get the Google Places API working. Testing the servers was also tricky, especially on Windows, where I had to enable the Telnet client to send and receive test messages. I also had to refine my logging function, where at first I had too much unnecessary outputs, which made debugging more difficult. Eventually, I was able to simplify the logs to focus on key events, which made it easier to track what was happening in the system.

8 Conclusion and Recommendation

In conclusion, I do recommend that *asyncio* is a suitable framework for an application server herd. It's great for handling many client connections and *I/O* tasks at the same time, which is exactly what we want in a server application. *Asyncio* works well with Python's built-in support for asynchronous code, making it easier to write clean and readable code. It also avoids the complications of multithreading while still allowing the server to respond to many requests efficiently. As long as we use the non-blocking libraries like *aiohttp*, *asyncio* can offer good performance and scalability for this kind of project.

References

Python Software Foundation. (n.d.). *asyncio — Asynchronous I/O*. In Python 3 documentation. https://docs.python.org/3/library/asyncio.html

Python Software Foundation. (n.d.). *Streams — asyncio stream support*. In Python 3 documentation. https://docs.python.org/3/library/asyncio-stream.html

aiohttp Contributors. (n.d.). *Client reference*. In aiohttp documentation. https://docs.aiohttp.org/en/stable/client_reference.html

Flexiple Editorial Team. (n.d.). *Python vs Java: What's the difference?*. Flexiple Blog. https://flexiple.com/compare/python-vs-java

GeeksforGeeks. (2023, July 19). *Garbage collection in Python*. https://www.geeksforgeeks.org/garbage-collection-python/

Notna, A. (2019, June 28). *Intro to async concurrency in Python and Node.js*. Medium. https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36

Node.js Foundation. (n.d.). *About Node.js*. https://nodejs.org/en/about