CS 131: Final Project Report
Implementing a proxy herd with asyncio

Carson Kim - Winter 2021

1 Abstract

This report will mainly discuss the pros of the `asyncio` library and its utility in an application server herd designed for rapidly-evolving data. I will discuss compatibility, ease of use, performance implications, and whether or not it is important to use the latest features offered by Python 3.9 or later. Moreover, the paper will discuss language-specific issues relating to Python - including type-checking, memory management, and multithreading. Lastly, I will be comparing the `asyncio` library with the Node.js approach.

2 Introduction

`asyncio` is Python's flagship library used for several foundational frameworks that allows concurrent, controlled execution over Python coroutines. In addition, it provides several high-level APIs for performing network input and output between server and client as well as the synchronization of concurrent code. These APIs proved to be rather intuitive when implementing a server herd for a Wikimedia-style service, and the rest of the paper will explore why I can give it a strong recommendation for such purposes.

3 Ease of Use

The `asyncio` library has intuitive, clean syntax. The typical way to declare coroutines is through the `await` and `async` keywords. These keywords appeared mainly in my server class, where the majority of the work was being done. This included declaring functions that handle client input as well as propagate messages as async functions, and the coroutines within them were declared with the `await` keyword.

According to the `asyncio` Python documentation, these coroutines are known as awaitables and can be accepted by the `asyncio.run()` function. This main function is responsible for managing the event loop and is the main point of entry for `asyncio` programs. I found this to work quite well with my program, as I included a function that "runs forever" and is where all important coroutines are executed.

4 Performance Implications / Compatibility

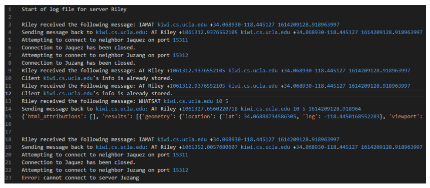
`asyncio` makes running concurrent, asynchronous code quite easy. It confers more power to the programmer in how they decide to control tasks / functions, and has stronger scheduling than the normal Python interpreter. Without `asyncio`, the Python interpreter could bleed resources by interrupting trivial tasks to switch to other threads.

`asyncio` seems to be quite compatible with the objectives of the server herd. Since we are required to make https requests, I did have to import the `aiohttp` module. This module is dependent on the `asyncio` library but makes it quite easy to run an event loop that declares a `ClientSession()` object that returns a response object that we can print and or manipulate. For these reasons, `asyncio` seems to achieve optimized, desired behavior.

5 Problems Encountered

The main difficulties of this project arose when propagating AT messages to other servers. First, there was the question of how this would be achieved. I decided that including a global dictionary that contained all of the server relationships could be useful here. Each server has a list of servers it can speak to, so naturally we should be able to traverse through this list and make sure the message is sent to each one. Of course, the function that handled this task was declared as asynchronous.

The other challenge was deciding when to propagate messages. There must be a condition under which servers do not propagate further, or else duplicate messages would be constantly bothering their server neighbors, resulting in an infinite loop. I did encounter this issue, and it ended up flooding my log files with repeated instances of the same request / action. These files ended up growing enormously large and eating up nearly all of my available disk space on my seas.net account.

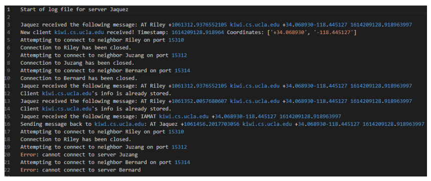


```
Start of log file for server Riley

Riley received the following message: IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Sending message back to kiwi.cs.ucla.edu: AT Riley +1061312.9376552105 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Attempting to connect to neighbor Jaquez on port 15311
Connection to Jaquez has been closed.
Attempting to connect to neighbor Juzang on port 15312
Connection to Juzang has been closed.
Riley received the following message: AT Riley +1061312.9376552105 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Client kiwi.cs.ucla.edu's info is already stored.
Riley received the following message: AT Riley +1061312.9376552105 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Client kiwi.cs.ucla.edu's info is already stored.
Riley received the following message: WHATSAT kiwi.cs.ucla.edu 10 5
Sending message back to kiwi.cs.ucla.edu: AT Riley +1061327.6560220718 kiwi.cs.ucla.edu 10 5 1614209128.918964
{'html_attributions': [], 'results': [{'geometry': {'location': {'lat': 34.06888734586305, 'lng': -118.4450168552283}, 'viewport':


Riley received the following message: IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Sending message back to kiwi.cs.ucla.edu: AT Riley +1061352.0057680607 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Attempting to connect to neighbor Jaquez on port 15311
Connection to Jaquez has been closed.
Attempting to connect to neighbor Juzang on port 15312
Error: cannot connect to server Juzang
```

Figure 1: Logging info stored for server Riley



```
Start of log file for server Jaquez

Jaquez received the following message: AT Riley +1061312.9376552105 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
New client kiwi.cs.ucla.edu received! Timestamp: 1614209128.918964 Coordinates: ['+34.068930', '-118.445127']
Attempting to connect to neighbor Riley on port 15310
Connection to Riley has been closed.
Attempting to connect to neighbor Juzang on port 15312
Connection to Juzang has been closed.
Attempting to connect to neighbor Bernard on port 15314
Connection to Bernard has been closed.
Jaquez received the following message: AT Riley +1061312.9376552105 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Client kiwi.cs.ucla.edu's info is already stored.
Jaquez received the following message: AT Riley +1061352.0057680607 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Client kiwi.cs.ucla.edu's info is already stored.
Jaquez received the following message: IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Sending message back to kiwi.cs.ucla.edu: AT Jaquez +1061456.2017703056 kiwi.cs.ucla.edu +34.068930-118.445127 1614209128.918963997
Attempting to connect to neighbor Riley on port 15310
Connection to Riley has been closed.
Attempting to connect to neighbor Juzang on port 15312
Error: cannot connect to server Juzang
Attempting to connect to neighbor Bernard on port 15314
Error: cannot connect to server Bernard
```

Figure 2: Logging info stored for server Jaquez

This spawned a number of problems, but it certainly taught me to be more careful in writing smarter, loop-free code.

6 Overall Design

The overall design of my program relied on several `async def` methods within my server class that would handle client input, retrieve the JSON response via the Google API search request, and propagate the messages to its nearby servers. The non-asynchronous functions were responsible for parsing and validating the arguments sent by the client. Of course, the server class also contains private variables that store the most recent coordinate info for every client as well as its timestamp. This is to ensure that servers always receive the most up-to-date information so they can handle WHATSAT requests properly.

Logging was done via Python's `logging` module. As required by the project spec, the log keeps track of all messages received by the server as well as its responses. The log also records any attempted connections to neighboring servers and alerts the user when an attempt fails. Figures 1 and 2 show brief snippets of Riley and Bernard's server logs when prompted with some simple requests mentioned in the spec. I intentionally closed connections for Juzang and Bernard to demonstrate that the other servers record any known outages yet still propagate to their other neighbors if possible.

7 Python vs. Java

This section will briefly discuss the advantages / disadvantages of a Python-based approach over Java in the implementation of this project.

7.1 Type Checking

Python's type checking system proved to be quite beneficial when working on this project, as it saved a lot of unnecessary programming hassle. Since Python is a dynamically typed language, there was no need to declare `asyncio` objects or coroutines as a certain type. A Java approach would likely have resulted in more verbose code that would have to accommodate to its static type checking requirements. I did not run into any runtime errors that resulted from type misinterpretations, and Python made it easy to write fast, fluid code that included a module with which I was completely unfamiliar.

7.2 Memory Management

Python utilizes an approach to garbage collection known as reference counting. Oftentimes, reference counting can be slower than Java's approach which involves identifying dead areas of code. In addition, Java's memory manager only allocates memory to objects and its garbage collector frees null pointers. It seems that since that Python's reliance on references to clean up objects could lead to worsened performance than Java, which has a much more robust and "smarter" way of determining dead code.

7.3 Multithreading

Since Python is an interpreted language, it naturally has an unconventional way of thread monitoring compared to its compile-time counterparts. The CPython interpreter doesn't even support multithreading, and the global interpreter lock is only responsible for isolated thread execution without actually providing any lock mechanisms to guarantee there aren't data races. Java offers keywords such as `synchronized` and `volatile` to help deal with shared resources in a program. The Java Memory Model's allowance of multithreading could potentially lead to increased risks in data races. For this project, it seems like Python should suffice as only one thread is required to run anyway.

8 Asyncio vs. Node.js

Node.js relies on callbacks to accomplish asynchronous behavior, but it appears as if it is quite similar to `asyncio` in their approach and design. Both can utilize the aforementioned awaitables that take in coroutines, and both run a main event loop. Such asynchronous functions are key to both `asyncio` and Node.js. Node.js can require some more advanced control flow to handle more complicated asynchronous tasks, but overall is rather competitive when compared to an `asyncio` approach.

9 Reliance on Latest Python Features

The release of Python 3.9 doesn't seem like it would disrupt a programmer's use of old features. According to recent documentation, most updates involve updates to event loop handlers. The `asyncio.run()` function would be the only major impact, as it now relies on coroutines to accomplish desired behavior. These coroutines, as mentioned earlier in the report, are declared with the async def keywords. There are other options available for implementing event loops, however. Namely, the `new_event_loop()` function which creates a new event loop that can do most of the work.

10 Conclusion

Overall, this project provided useful insight into the advantages of Python's `asyncio` networking library. Due to its intuitive methods of running event loops and handling coroutines, I think the module is a strong candidate for replacing the Wikimedia platform for my application. It works seamlessly with not only TCP requests but also HTTP requests, with great help from the `aiohttp` library. Python's slick type checking system also allows for fluid, cleaner code when working with this particular module. Through this experience, I can guarantee that any application developed with `asyncio` functions is maintainable and reliable.

11 References

https://docs.python.org/3/whatsnew/3.9.html#asyncio
https://docs.aiohttp.org/en/stable/client_quickstart.html
https://eng.paxos.com/python-3s-killer-feature-asyncio
https://towardsdatascience.com/a-better-way-for-asynchronous-programming-asyncio-over-multi-threading-3457d82b3295
https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36