CS 131 - Project Report
Proxy Herd with `asyncio`

Abstract

This paper focuses on Python's `asyncio` library and whether it is a suitable platform for implementing a proxy herd; first part is dedicated to a brief explanation of the project specifications and the required functionalities to be implemented using `asyncio`. The next part of the paper is directed towards analyzing the suitability of `asyncio` for similar applications, including how easy it is to use and its performance implications. We will then focus on the pros and cons of using this library, explaining all the problems that were faced while implementing the proxy herd program, and finally we will compare different aspects of Python, such as type checking, memory management, and multithreading to that of Java. The last part of the paper includes a brief comparison between the approaches of Python and Node.JS.

1 Introduction

The project is centered around the idea of implementing a proxy herd using the `asyncio` library. The Python program can be executed using one of the five possible server names as its argument, and each execution causes the initiation of a proxy server with the specified name. These servers are able to "talk to" each other using TCP connections to communicate the location data they receive from moving clients. Each client can either send its location and time to a server or request a list of places in a certain radius of a different client; this aspect uses the Google Places API to receive the correct places in `json` format. A flooding algorithm is implemented so that when each server receives a location update, it can propagate it to all the servers it is connected to, so that eventually all the servers in the network can keep a record of the most recent location of each client.

2 What Is `asyncio`?

`asyncio` is a library provided for Python language to write concurrent programs. An asynchronous program is one that makes use of `async/await` to let another task execute while it is waiting for a different task to finish. This is different than parallelism, since it is like two threads that take turns executing rather than both executing simultaneously. `asyncio` provides the users with a set of high-level and low-level APIs from which I made extensive use of `Streams` which are "async/await-ready primitives to work with network connections"[1].

3 Suitability of `asyncio`

3.1 Ease of Use

The most time-consuming part of the project for me was learning about asynchronous operations and reading up on the `asyncio` library. The library documentation is well-written and clear to follow. However, not having used Python in a relatively larger program before and not being familiar with asynchronous operations in general caused me to have to search elsewhere for many of the keywords used in the documentation. However, after the research and the readings, writing the code wasn't very difficult. The syntax and the way to use each method is easy to understand.

3.2 Performance Implications

`asyncio` is a perfect fit for I/O-bound applications, according to the documentation. Our program runs in a single thread and is constantly waiting to receive data from clients, so it is an I/O-bound application; therefore `asyncio` must be a good fit for it. I haven't taken any performance measurements, but I haven't noticed any slow operations in the program; this might change if instead of 5 we had tens or hundreds of servers and much

more data was transferred between them and clients, but `asyncio` seems to be doing a good job for a small application like this. Another bright side is that by using `asyncio` we have avoided the complexity of using multithreading and made our jobs a lot easier.

3.3 Pros of `asyncio`

After learning the library and becoming comfortable with the Python language itself, `asyncio` was relatively easy to use. The fact that it makes use of concurrency rather than parallelism is helpful since we don't have to deal with synchronization complexities. And I found that there was a built-in method for any operation that I needed for this specific project so that made my job, as a programmer, easier.

3.4 Cons of `asyncio`

The time that was spent on learning the library and its methods was quite significant. Another issue is the disconnecting of the servers. Since each server stores the information about clients' locations on a hash table, when servers die the information dies with them. If there was a central, shared database for all the servers this wouldn't be an issue.

4 Python vs. Java Approaches

4.1 Type Checking

"Python is a dynamically typed language. This means that the Python interpreter does type checking only as code runs, and that the type of a variable is allowed to change over its lifetime" [2]. Java, on the other hand, is a statically typed language. When introducing new variables we must explicitly declare their types and the type cannot change over the scope of that variable. Type errors in Java are catched at compile time [3]. Although dynamic type checking seems to be better since we don't have to worry about declaring types of variables, it could slow the program down in larger applications. The reason is that "all values that occur when a program executes, need to [become] tagged with a type, [and] maintaining this runtime type information is expensive[4]". Other than the slow down, Python's dynamic type checking bring unreliability to our application; since the errors are only caught at runtime, a server that was expected to go online could be stopped from doing so due to a type error.

4.2 Memory Management

Both Python and Java use garbage collectors so that the users don't need to deal with explicitly freeing allocated memory. They do this using different methods; Python uses reference counting which can end up being a more expensive operation than Java's methods. In both cases there is no explicit allocation of memory, either. I don't believe the difference between their garbage collection methods has a huge impact on the proxy herd application, since we don't deal with a large amount of memory allocation and deallocation.[5]

4.3 Multithreading

I personally don't have experience with Java multithreading more than what we did for the "Java shared memory performance races" homework and so the comparison between to the two language based on personal experience wouldn't be the most accurate comparison between the two languages. It seems to me that using Java multithreading is quite straightforward. Figuring out how to use the `asyncio` library was much more time consuming than figuring out how to implement a multithreaded program in Java.

5 `asyncio` vs. Node.js

Similar to `asyncio`, Node.js is an event-driven, I/O based model. Node.js is a relatively new language comparing to Python, and because of this, the documentation and its community for asking questions is limited. Node.js was not designed for intensive CPU use and so is probably not a good platform for a large application with lots of data processing. However, if we want to implement our proxy herd application on mobile devices (for example for the clients' use), Node.js is the way to go since Python is not well-supported in mobile devices.

6 Conclusion

Overall I believe that `asyncio` was a good platform to use for the application we wanted. However, using this approach for Wikimedia might not be a great idea; unless if we are able to use a database that is shared between all the servers to save the information so that the overhead due to the flooding algorithm wouldn't be a problem anymore.

Notes

[1] https://docs.python.org/3/library/asyncio.html
[2] https://realpython.com/python-type-checking/type-systems
[3] http://www.cs.pomona.edu/ kim/CSC051GF14/lectures/Lecture32/Lecture32$_3$*.html*

[4]Webber, Adam Brooks. Modern Programming Languages: a Practical Introduction. Franklin, Beedle Associates, 2011.

[5]https://docs.python.org/3/c-api/memory.html