CS 131 Python Project
Proxy Herd with `asyncio`

Rowan McKereghan
University of California Los Angeles

Prof. Paul Eggert
Discussion 1B

Abstract

In this project, I implemented a proxy herd prototype with `asyncio`, an asynchronous networking library, and did research into whether that library was suitable for a real-life, full-size application, as opposed to other languages' approaches including Node.js and Java.

1 Introduction and Prototype Overview

To analyze whether or not the application server herd architecture was best implemented through Python's `asyncio` library, my prototype allowed for a number of different messages and queries both to and from clients and servers. Servers responded to `IAMAT <ClientName> <ISO 6709 Location> <POSIX Time Stamp>` messages with `AT <ServerName> <POSIX Time Difference> <ClientName> <ISO 6709 Location> <IAMAT POSIX Time Stamp>`. They then propagated these `AT` messages to other neighboring servers through a simple flooding algorithm, which allowed the spread of the most recent location information of every client who sent an `IAMAT` message. This is so that when any query in the format `WHATSAT <ClientName> <radius> <numResults>` was posed to any server, they would all be able to respond provided one of them had received an `IAMAT` prior. The server would then respond to the `WHATSAT` query with a copy of the `AT` message response and a JSON object received from the Google Places API that contained the results for any clients within the desired bounds. A difficulty I encountered was formatting my URL for a JSON request correctly. After checking multiple official Google formatting description sites, I figured out how to not get an `INVALID REQUEST` or a bad API key error. Another difficulty I encountered was figuring out how to using Python's `pdb` debugger to find what was going wrong with my code. Making sure my input format checkers were correct my using print statements only got me so far, so I had to do some research into `pdb` and figure out how to use it to correct my checks.

2 Comparing Python and Java

Here we will compare the Pythonic way of type checking, memory management, and multithreading to Java, and also bring `asyncio` into the discussion as to the maintainability and reliability of larger implementations of applications involving architectures like the above prototype.

2.1 Garbage Collection

Python's garbage collector implementation is pretty simple: as soon as there are no more references to an object, it is freed.[1] This method is pretty fast in keeping track of which objects can be freed or not, but it has takes a performance hit when dealing with cyclical data structures. Java, on the other hand, uses an incremental mark-and-sweep method to free unreachable objects. The mark-and-sweep algorithm[2] traverses through every object that it can reach in the object tree and marks them, then frees all unmarked (i.e., unreachable) objects. Java only calls this method in a fixed amount periodically as to not impact performance as much as doing a full sweep every time an object needs freeing. While Java's method is more effective against cleaning up cyclical data structures, there in fact aren't any of these in a potential server application herd implementation, making Python a good fit in this regard due to its simplicity.

2.2 Multithreading

Another downside of Python's garbage collection system is that it necessitates the Global Interpreter Lock,[3] a mutex that only allows threads to access Python bytecode one at a time. If multiple threads were run in parallel, the reference count method for garbage collection could potentially result in undefined behavior or memory leaks. Java allows for true parallel

[1] Explained in Prof. Eggert's lectures

[2] The mark-and-sweep algorithm: https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/

[3] https://wiki.python.org/moin/GlobalInterpreterLock

multithreading with `synchronized` and thread Executor objects as safety checks, but also allows for risks that could potentially result in race conditions if code is written poorly. In short, Python chose single-threaded simplicity and Java chose fast, parallel performance with more complex code and higher risks. In the case of our prototype, multithreading wouldn't necessarily speed up the process, as waiting for server responses would take about the same time if single-threaded or multi-threaded. However, if at some point our servers had to read and write a large amount of data, a multithreaded reader and writer would potentially speed up our program a lot.

2.3 Type Checking

Since Python is a dynamically-typed language, it can assign any legal value to any legal variable name that a developer can create.[4] This has its advantages: declarations are shorter, Python is far more readable, and any variable can be passed as an argument into any function. However, this makes Python errors a bit harder to debug. If a developer mislabels a variable, or writes some undefined behavior, Python could potentially still compile and run despite having errors that would've been caught in a statically-typed language like Java. Java makes sure that all variable values match their declared types at compile time, and if they don't, it returns an error. While this makes Java code more complex and lengthy, it is also much safer than Python's method. For our project, either method works, as we assume that professional developers know their way around both systems.

3 Comparing `asyncio` and `Node.js`

`asyncio` and `Node.js` are quite similar in the sense that they are both libraries that support asynchronous single-threaded event-loop-based concurrent programming. Python and JavaScript are also both simpler languages than many of their peers. However, they do have their differences: `Node.js` still provides support for multithreaded parallelism with multiple cores,[5] and `asyncio` uses coroutines instead of callbacks like `Node.js`.[6]

4 Language-Related Benefits and Detriments of `asyncio`

In this section we will discuss the pros and cons of `asyncio` as related to writing, performance, and backwards compatibility.

4.1 Ease of Writing

It is in fact quite easy to write and run server-herd applications with `asyncio`. Functions like `open_connection()` and `start_server()` make it very simple to connect servers to ports and each other, especially with Python's dynamic type checking environment made for speeding up developer time. I myself had more trouble with JSON libraries and debugging my non-asynchronous code than I did with `asyncio` methods. This is definitely a benefit of using `asyncio`.

4.2 Performance Implications

`asyncio` is single-threaded, meaning it will always be slower on multicore computers that use multithreading to speed up time intensive processes. As described before, multithreaded reading and writing would significantly speed up the implementation of our servers if given large inputs and outputs to process. The fact that we must rely on single-threaded performance of potentially massive tasks once our prototype is built to scale could result in very slow communication between clients and servers. This is a detriment to using `asyncio` for our server herd implementation.

4.3 Reliance on Newer Versions

While functions like `run()` and the `-m` option are very helpful for automatically executing coroutines, managing event loop queues, and executing commands for Python, respectively,[7] they aren't necessary for implementing these features. All of the methods mentioned above have workarounds, like combining `new_event_loop()` and `run_until_complete()` for `run()`, and using Replit[8] instead of the `-m` option. Since coding with backwards compatibility is possible, I would say the newer versions of `asyncio` are solely for developer benefit, and thus make it a net positive.

5 Recommendation

Before I make my recommendation, there are a couple more pros and cons to using `asyncio` that should be mentioned. First, Python has many, many libraries that can be used in tandem with `asyncio`. In my prototype, for example, `aiohttp` and `argparse` were used to make implementing JSON requests and parsing command line options and arguments much easier. Since `asyncio` can't do everything as well as other libraries that could potentially implement a server herd, taking advantage of other Python libraries gives it a major benefit. On the other hand, since asynchronous code executes in random order, we cannot make as many assumptions about when messages sent to servers will be processed. It could

[4] https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html

[5] https://nodejs.org/en/about/

[6] https://docs.python.org/3/library/asyncio.html

[7] https://docs.python.org/3/whatsnew/3.9.html

[8] https://replit.com/languages/python3

potentially take a very large amount of time to process a to-scale `WHATSAT` JSON request, to the point where by the time it finishes another, more relevent `AT` or `IAMAT` message has already been received. This is definitely a downside of working with `asyncio`. However, all things considered, I would have to recommend using `asyncio` to implement an asynchronous proxy server herd, simply because its ease of use and backwards compatibility outweigh the performance benefits of multithreaded server implementations, as reading and writing-intensive single tasks wouldn't necessarily occur that often as opposed to many, small reading and writing tasks. `asyncio` is a good option for a larger, multi-server application.