Evaluating A Proxy Herd with an asyncio implementation: Performance and Suitability

Stefanie Shidoosh – University of California, Los Angeles

Abstract

This project's goal is to implement a proxy herd using the Python asyncio library, utilizing a flooding algorithm. As I implemented the specifications, I am to assess whether or not asyncio is a suitable library for this sort of application. The following research will compare and contrasts pros and cons of the library, while justifying my position on the matter. I will also touch on the trouble I ran into during my implementation, and address the concerns of using Python for this implementation rather than Java, such as Python's type checking, memory management, and multithreading performance relative to Java's handling of these issues. By comparing the Python asyncio library to that of Node.js, I will delve into these issues and continue to compare the two languages within the context of this model application.

Introduction

This project is inspired by the LAMP-stack architecture that Wikimedia utilizes, such that we want our design to deal with the bottleneck such an application implemented with LAMP would face, say, if the clients were more mobile and updates happen more frequently. In its implementation, clients could connect to servers that will serve their needs best, and within this project's scope, this implies the server nearest and then that server communicates with the others and updates accordingly.

Thus, to address the bottleneck, the project specifications suggest the Python asyncio library. Asyncio proved to make life easier, such that the implementation of the proxy application server herd benefits from the library's direct dealing with the aforementioned bottleneck. By dealing with connections asynchronously (hence, the name of the library) via a flooding algorithms allows for a large amount of connections that can update servers which then communicate, again via the flooding algorithm. In this way, the transmission of significant data and potentially a significant amount of data is facilitated by asyncio. However, for the implementation design the aiohttp library is required to manually create and send an HTTP GET, as ayncio only support the TCP and SSL protocols. Though one could consider this a downside, it is but a minor inconvenience considering asyncio's power and the availability and easily understandable interface of the aiohttp library. We also need the json library to get the Google Places API information and package accordingly.

1. Design and Specifications

This project requires that the proxy application server herd handles connections asynchronously, such that the data transmitted via these connections are handled efficiently.
There are three commands that the severs send to clients: IAMAT, WHATSAT, and AT. The client connection to the proxy server herd is a TCP connection that is pushed onto an event loop, with the loop processing inputs from servers sequentially in the format of the three aforementioned commands.

1.1. IAMAT Command

This command tells the proxy server herd where a client, specified by its ID, is. The client ID can be any string of non-white-space characters.

It has the following operands: client ID, latitude and longitude in decimal degrees using ISO 6709 notation, POSIX time when the message was sent. Each operand is separated by one or more white space characters.

The servers then respond with an AT message of the form with the command, the server ID that got the message, the difference between the server's idea of when it got the message from the client and the client's time stamp, latitude and longitude in decimal degrees using ISO 6709 notation, POSIX time when the message was sent.

1.2 WHATSAT Command

This command allows for clients to query for information about places near other clients' locations, with a query using this format: the name of another client, a radius from the client, and an upper bound on the amount of information to receive from Places data within that radius of the client. The radius must be less than or equal to fifty kilometers, and the upper bound can be no more than twenty items.

The server then responds with an AT message, which is in the format of a JSON-format message that packages the format of a Google Places Nearby Search request.

1.3 AT Command

This command allows for servers to communicate with each other. The AT commands output is propagated among the servers via the flooding algorithm such that locations is the only field communicated. Place information should only be retrieved through the Google Places API, rather than between servers when requested.

2. Asyncio suitability

Per its documentation, the asyncio library "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," offering event loops and co-routines that allow for writing "concurrent code in a sequential fashion" based on yield from [1].

Because of such behavior, I argue in favor of asyncio for our purposes of implementing our proxy herd. The biggest setback and most intimidating aspect of the project was how to tackle the issue of handling client-server communications, and how to propagate the data among the servers. Thus, this module comes to the rescue with Task objects that package requests from clients via a co-routine object. The event loop will run to completion such that all Tasks must be finished executing and exiting the client connection, while the servers receive the corresponding messages and communicate according, again, until completion.

This functionality works great for the behavior we want – asyncio event loops schedule co-routines in a sequential fashion. Data can then be transmitted via a TCP connection, one of the functionalities asyncio is built for [1], and can be processed asynchronously. Servers maintain their own buffers for their corresponding messages, and thus the communication between the servers cannot be muddled with as data is transmitted between the servers, as demonstrated in the logs generated from proxy server heard implementation. Thus, the advantage of handling asyncio Tasks in this efficient manner proves for a relatively painless implementation.

3. Problems

As stated prior, there was not many issues I ran into while implementing the proxy server herd with asyncio, as the module seems built for this sort of desire functionality of propagating information among servers with specified relationships between each other. However, the most trouble I had was the sheer lack of experience with communicating with an external API such as Google Places. Asyncio is not equipped with HTTP GET, as mentioned before, thus this was one issue I would have not been able to handle if I depended solely on asyncio, since it only provides support for TCP, UDP, and SSL [1].

3. Python-based Approach and Java-based Approach Comparison

Python's approach to memory management and multithreading relative to Java's approach are points of major concerns when considering one design implementation over the other, and with these two aspects, Java proves to be winner of the two. However, considering the asyncio capabilities and Python's duck typing, Python offers a better approach to type-checking as it free a programmer of meticulously declaring types, and instead getting right down to business.

3.1 Memory Management

The Java garbage collector proves to be much more effective than Python's memory management model, and thus concern in this regard is valid.

The difference between the memory models boil down to the garbage collector implementation of each, with reference-count based garbage collection and generational garbage collection for Python and Java, respectively.

Reference counting frees objects as soon as they have no references, and thus most of the time garbage collection is rarely called to clean up. However, reference-count based garbage collection introduces overhead that Java's generational garbage collection does not.

Java's mark-and-sweep approach allows for blocks of memory in their frequency of generation to be less likely freed. This requires dual-assessment of the heap, but because of Java's advantage of headedness, this garbage collection occurs on its own thread and thus efficiency is maintained.

3.2 Multithreading

Quite simply, Python is not suited for multithreading. Because it is an interpreted language, the ability for concurrency is frankly not viable as lines of code must be executed sequentially and threads cannot communicate via processes as they are being executed, again, line by line.

Java, however, does allow for such parallelizability and thus relative to Python's lack of such functionality, is a valid concern when considering the two approaches. In regards to our proxy server herd context, the performance is indeed hindered when comparing to that of a Java style approach. Servers in the Java approach could be updated concurrently, whereas with the Python approach, such a feature is not available and thus efficiency is not as optimized as it could be.

3.3 Type-checking

Python's use of duck typing facilitates asyncio's use of asynchronous co-routines. When data is communicated following a yield, the code does not need to specify what is the type of the object that one would expect. This allows for flexibility, and ultimately faster development time. Thus, prototyping is a catalyzed process, whereas the Java developer has to agonize over what an object type will be and handle cases where the object is not the type that it was expected to be – the Python approach does not have to concern itself with this feat [3].

4. asyncio vs Node.js

Per its documentation Node.js is an asynchronous, "event-driven, non-blocking I/O model," which proves to be very similar to our beloved asyncio library.

Processes are scheduled as callbacks, which are analogous to asyncio co-routines, onto the Node.js event loop, and are processed until they yield. The event loop, analogous to asyncio event loops, then awaits input to continue processing [4].

Node.js offers a more streamlined approach for frontend and backend development. It is a JavaScropt runtime and thus offers a compatibility for application implemented with this scripting language, especially considering these applications would depend on both client and server sides. However, asyncio is better suited for dealing with larger client-server traffic, equipped with handling larger amounts of data.

References

[1] "18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks¶." *18.5. Asyncio - Asynchronous I/O, Event Loop, Coroutines and Tasks - Python 3.6.4 Documentation*, Python Software Foundation, 10 Mar. 2018, docs.python.org/3/library/asyncio.html.
Note: See 18.5.9

[2] Sehar, Uroosa. "Java vs. Python: Which One Is Best for You?" *Application Performance Monitoring Blog*, AppDynamics, 21 Mar. 2017, blog.appdynamics.com/engineering/java-vs-python-which-one-is-best-for-you/.

[3] Sommers, Bill Venners with Frank. *Strong versus Weak Typing*, Artima, 10 Feb. 2003, www.artima.com/intv/strongweak.html/.

[4] Saba, Sahand. "Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js." *Math Code by Sahand Saba Full Atom*, 10 Oct. 2014, sahandsaba.com/understanding-asyncio-node-js-