Proxy Herd with asyncio

Rishab Khurana

University of California, Los Angeles

Abstract

This paper will discuss some of the advantages and disadvantages of development using Python's ayncio for server heard applications. In particular, we will discuss an implementation of an asyncio application to create a faster Wikimedia Platform meant for mobile devices. We will discuss the nature of developing in Python with asyncio in comparison with other platforms and give a final recommendation on whether or not it is an impressive development environment for the mentioned application.

1 Introduction

1.1 Wikimedia and Server Herd

The Wikimedia server platform is the foundation for several websites, including the well-known Wikipedia. The platform utilizes several applications: Linux, Apache, MariaDB (database), and source code in PHP + JavaScript. It is a very well-integrated platform with coherent diagrams that explain its functionality well (Wikimedia, 2024). Wikimedia utilizes a core database and caches so that other server's interact with each other by referencing and editing the core database and caches. This can cause some response time delays because if a user u uploads their location to server A, but server B must wait for server A to upload their data to MariaDB before learning of the user u's location. This bottleneck is heavily explored in this paper via the implementation of an application server herd. Rather than solely relying on communication through the core database and caches, server herd take advantage of direct communication between servers to speed up the propagation of client information.

1.2 Intro to Python's asyncio?

Asynchronous I/O (asyncio) is a Python library that is used to write concurrent code in Python. It is generally a foundation for "Python asynchronous frameworks that provide high-performace network and web-servers, database connection libraries, distributed task queues, etc." (asyncio documentation, 2024). It is often a "perfect fit for IO-bound and high-level structured network code" (asyncio documentation, 2024). Using the async and await keywords, one can define asynchronous methods that can be awaited within other asynchronous methods. This process is very easy to do, and it makes server development a lot easier (than, say, PHP + JS). For instance, a file could start a python server by awaiting for asyncio.start_server(), and then that server can open a connection with another server by awaiting for asyncio.open_connection(). Then, the server can communicate with the new connection by using the instances of the StreamReader and StreamWriter objects returned by asyncio.open_connection(). The await keyword is important here because it tells Python to wait for the connection to open first before starting any communication. Thus, on the surface level, asyncio is a versatile Python library that can allow seamless development of network applications.

2 Server Herd Application

To explore Python's asyncio and its potential versatility, I made an application that is a proxy for the Google Places API that makes use of a server herd. It has three main features. A client can tell one of five servers where they are through command IAMAT. The server then propagates this information via a flooding algorithm to the other servers using AT commands – this feature highlights the server herd functionality of this application. Finally, users can query clients to see nearby locations to the queried client using a WHATSAT command. Each of these features utilize asyncio for two lines of communication: server to client, and server to server.

2.1 `IAMAT`

This command allows clients to tell the servers where they are. The input is of the following form: `"IAMAT {client id} {coordinates} {time of send}"`, where the bracketed parts are user input. The client id is custom (according to the client) – for example, I can have rishik50016 as my client id. The coordinates are the coordinates of the client; the format for the coordinates must follow: `"[+-]lat[+-]long"` (`[+-]` is RegEx for + or -), and lat and long are the lattitude and longitude of the client. The time of send is the time the client sends the information to the server in POSIX time (time of non-leap seconds since Janurary 1st, 1970). The server then responds to this message with an `"AT {server name} {time diff} {client id} {coordinates} {time of send}"` indicating to the client that the server `server name` received the message `time diff` seconds after the client the message. The server then records the the `AT` message in its database and propagates this message using inter-server communication (see below section).

2.2 `AT`

Once a server sends an `AT` message (format displayed in last section) to the client, it also sends it to their friends. A server's friends are predefined in the program, but a server is not friends with all other servers – only a strict subset of the servers. Here is the snippet of code from my program that defines a server's friends:

```
FRIENDS = {
    "Bailey":["Bona", "Campbell"],
    "Bona":["Bailey", "Clark", "Campbell"],
    "Campbell":["Bailey", "Bona", "Jaquez"],
    "Clark":["Jaquez", "Bona"],
    "Jaquez":["Clark", "Campbell"]
}
```

You can see that each of the servers maps to an array of other servers – those are the server's friends (and friendship is bidirectional). Once a server S receives an `AT` message, it does the following: S first checks if the `AT` message contains database info that is more updated than S's database; if it is, then S will update its database and propagate the message to S's friends; if it isn't, then S will do nothing with the message. This is because if S receives more updated info, then S will learn of it and S will give it to the friends of S, but if S doesn't receive more updated info, then S – and as a consequence, all of S's friends – will have no need for the message. Thus, this algorithm always propagates the most updated client info throughout the servers via direct server communication.

2.3 `WHATSAT`

The `WHATSAT` message allows clients to see nearby locations to other clients (including themselves). It takes the form `WHATSAT {client id} {radius} {info bound}`. The `client id` specifies which client to give information about. The `radius` specifies how far to look from the `cliend id`'s location, and the `info bound` specifies how an upper bound on how many locations to return to the person querying for information. Once a server S is given a `WHATSAT` command, S will query its database using `client id` to find the location corresponding to `client id`. Then using this location along with the `radius`, S uses `aiohttp` to make an API request to the Google Places API to get a JSON describing all the locations within `radius` of `client id`'s location. Then, the server informs the client (doing the querying) of the first `info bound` (or less if there are not `info bound` locations) locations.

2.4 Problems in Development

Many of the issues I had in development surrounded my knowledge and research with `asyncio`. Once I had a good grasp of how to use `asyncio`, a lot of the project came pretty naturally to me. One issue was involved with the loop of the server. The intended behavior is for the five servers to constantly take client input in and respond accordingly; after each server response, the server would then take more input in. Thus, the server must constantly be reading input. The issue I had is that the server shut down after one command from the client. My initial fix to this was to have the server run a `while True` loop for reading user input (i.e. While True, read the next line). But this caused issues when implementing inter-server communication using `AT` because sending messages to servers essentially count as commands (just instead of from a client, its from a server). But, after some research, I cam across the `StreamReader.at_eof()` method, which essentially returns true only if the client-side buffer is empty. This was more compatible with inter-server communication as well, and the server worked nicely once I replaced `while True` with `while (not (reader.at_eof()))`. Another notable issue I ran into had to do with querying the google API. I ran into some issues with filling out the parameters for the query URL such as formatting of the coordinates, where to input the API key, and where to put the radius. A lot of this was also solved with sufficient research into the Nearby Places documentation, where parameter formats were stated very clearly. Getting started with the project as a whole took some time, as reading through the documentation was a considerable amount of work, but I found that understanding the functionality came naturally, and the `asyncio` documentation was very good in helping users understand how things worked; however, a decent amount of self-experimentation was still needed to understand the library enough to make the server herd application.

3 Strengths and Weaknesses of Python and `asyncio`

3.1 Strengths

Development in `asyncio` was very seemless and fast. The documentation was nice, and I picked up on many important concepts very fast. The ability to use `await` and `async` allows for a creation of a server in moments. Given that the Wikimedia Platform utilizes PHP and JS, creating a server in Wikimedia can take a considerable amount of time when compared to `asyncio`. Another notable strength was the application I created. I was successfully able to create a server herd using `asyncio` within a very reasonable amount of time – this solves a big issue with the Wikimedia platform. With the server herd, the servers no longer has to interact with other servers using a central database and communication occurs directly between servers using the `AT` command. This is nice for a mobilized version of the Wikimedia platform. The asyncio library allows us to model threads (concurrent programming), which is also a strength because it supports timely I/O operations. In terms of performance, Python has the advantage of being easier to optimize than JS or C/C++ alternatives because of the easier and more readable development environment in Python. Python voids many of the common unsafe issues associated with C/C++ because it has its own garbage collector and makes development easier. Python is also easier to learn and understand compared to JS, making the development process cleaner. Moreover, the server herd application in `asyncio` allows for faster response times because servers aren't forced into indirect communication. Thus, although Python is a slower (interpreted) language, the features of `asyncio` allow optimizations that are hard to implement in PHP + JS. In Python 3.9+, `asyncio` added several features (such as `asyncio.run()` which supports easier development of servers (and server herds). For example, a developer can create a method that initializes a server, and all the developer needs to do to run the server is call `asyncio.run()` on the server. And testing is easy because developers can use `python3 -m asyncio` (for later versions) in order to get an interactive environment where they can test `async` operations.

3.2 Weaknesses

Despit `asyncio` allowing for nice features to develop with in Python, the language (Python) itself is a lot slower compared to JS. This is mainly because Python is an interpreted language. Its dynamic nature forces runtime to be slower; dynamic type checking (variable types determined at runtime), the interpreter, large library support all force Python to be a slower language in comparison to JS or C/C++. Thus, if an `asyncio` alternative is implemented in other (compiled) languages, it may have the potential to be faster than what we see in Python. Additionally, Python can be viewed as a less reliable language due to it being an interpreted language – types, for example, can fail at runtime because they weren't checked at compile time (what JS, C/C++ does). Another issue with `asyncio` is concurrency vs. parallelism. `asyncio` only supports concurrent programming: it manages several instructions at one time (timesharing). Parallel programming has the advantage of *running* several instructions at one time. Although in a server I/O operations almost need to force a concurrent nature at times (waiting for I/O before continuing execution), having the option to execute certain operations in parallel definitely would increase speed and response time, but `asyncio` does not support this. Another issue with `asyncio` is that it does not support HTTP GET requests. This can be a problem because servers sometimes need to query using GET requests. For example, my server heard application utilized a GET request to query from Google Places API's Nearby Search to get locations around a client. For this, I could not use `asyncio` and needed to use `aiohttp`. Another issue has to do with features that are introduced in later versions. In the strengths section, we discussed `asyncio.run()` and `python3 -m asyncio`, and although these are nice features, the backwards compatibility could be frustrating. The older versions of python that do not support these features will definitely cause the program to fail, and lower level API for `asyncio` may need to be referenced to make a current `asyncio` program more backwards compatible. The lower-level API, however, would be much harder to develop in relative to current versions of `asyncio`.

4 `Node.js` as an Alternative

`Node.js` is an event-driven JavaScript runtime. Like the `asyncio` framework, it uses a concurrent model of execution rather than parallel by default. But, unlike `asyncio`, `Node.js` does offer the developer the option to have parallel execution, meaning there is flexibility from the developer side to potentially take advantage of parallel execution when they can. Moreover, `Node.js` does support HTTP GET requests, meaning an external library isn't needed (like `aiohttp` in Python) to perform GET requests. However, development in JavaScript can be more tedious than in Python, as Python is a very understandable language that has a vast amount of libraries to perform functions that could take several extra lines of code in other languages such as JS. However, given the greater support for parallel programming (greater scalability) and the ability to make use of GET requests, utilizing `Node.js` solves many of the issues with development using the `asyncio` framework. Thus, it is a strong alternative to development in `asyncio`.

5 Final Recommendation

All in all, I do recommend utilizing `asyncio` for development of the application server herd for a retweaked Wikimedia platform. Python is a language that is very easy to develop in and creating an application such as a server herd can be done in a short amount of time. I do believe it is worth it to further look into developing such an application using `Node.js` because of the potential speedups it has to offer. But, at the moment and given my current experience with both platforms, I think the advantages of development in Python outweigh its costs, and I believe that development in alternatives (such as Node.js or PHP + JS) can be cumbersome and unnecessarily extend development time.

6 References

Wikimedia Platform Description:
https://meta.wikimedia.org/wiki/Wikimedia_servers

Wikimedia Diagram:
https://wikitech.wikimedia.org/wiki/Wikimedia_infrastructure

asyncio documentation
https://docs.python.org/3/library/asyncio.html

Nearby Places query documentation
https://developers.google.com/maps/documentation/places/web-service/search-nearby

Node.js documentation
https://nodejs.org/docs/latest/api/