CS 131 Project Report

Ryan Riahi

March 2021

Abstract

In this report I look into Python's asyncio asynchronous networking library for building an application server herd. I will evaluate the pros and cons of using asyncio by building a proxy for the Google Places API with the asyncio library as a framework. I will then compare the pros and cons of aysncio with those of using Node.js to build this kind of application.

1 Introduction

We are building a new Wikimedia-style service designed for news, where (1) updates to articles will happen far more often, (2) access will be required via various protocols, not just HTTP or HTTPS, and (3) clients will tend to be more mobile. We are concerned that this new service's response time will be too slow because the PHP+JavaScript application server could be a bottleneck. I have been asked to look into a different architecture called an "application server herd", where the multiple application servers communicate directly to each other as well as via the core database and caches. The interserver communications are designed for rapidly-evolving data. We believe that Python's asyncio asynchronous networking library might be a good match for this problem, since asyncio's event-driven nature should allow an update to be processed and forwarded rapidly to other servers in the herd. As such, I have been delegated to look into asyncio, specifically, how easy is it to write applications using asyncio, how maintainable and reliable those applications will be, and how well one can glue together new applications to existing ones. My boss suggested that writing a proxy Google Places API using asyncio will give me a good insight into the library and its pros and cons.

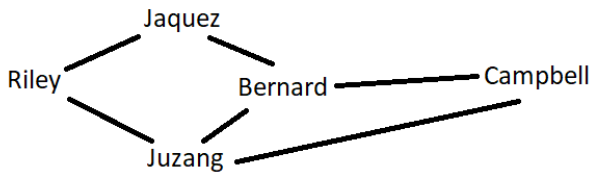
2 Application Server Herd

In this section I will go over the details of my proxy Google Place API and how I used Python and asyncio to build it.

2.1 Server Design

My application consists of five servers named Riley, Jaquez, Juzang, Campbell, and Bernard. These servers communicate to each other bidirectionally based off the pattern in figure 1.

Figure 1: Bidirectional Communication Link



The servers communicate client time and location updates to each other using a simple flooding algorithm. I used simple and easy Python dictionaries to assign each server to a given port and to their neighboring servers with whom they communicate. Each server also accepts TCP connections from clients that emulate mobile devices with IP addresses and DNS names. Servers will respond to all invalid command with a '?' followed by the command Lastly, the

servers will all log every input and output file named "serverName".log.

2.2 Client Message: IAMAT

A client should be able to send its location to the server by sending a message using this format:

```
IAMAT riahi +34.068-118.445 161420.2
```

The first field IAMAT is name of the command where the client tells the server where it is. Its operands are the client ID (in this case, riahi), the latitude and longitude in decimal degrees using ISO 6709 notation, and the client's idea of when it sent the message, expressed in POSIX time. A client ID may be any string of non-white-space characters. (A white space character is space, tab, carriage return, newline, form feed, or vertical tab.) Fields are separated by one or more white space characters and do not contain white space; ignore any leading or trailing white space on the line. The server should respond to clients with a message using this format:

```
AT Riley +0.263 riahi +34.068-118.445 161420
```

where AT is the name of the response, Riley is the ID of the server that got the message from the client, +0.263 is the difference between the server's idea of when it got the message from the client and the client's time stamp, and the remaining fields are a copy of the IAMAT data. In this example (the normal case), the server's time stamp is greater than the client's so the difference is positive, but it might be negative if there was enough clock skew in that direction.

2.3 Client Query: WHATSAT

Clients can also query for information about places near other clients' locations, with a query using this format:

```
WHATSAT riahi 10 5
```

The arguments to a WHATSAT message are the name of another client (e.g., riahi), a radius (in kilometers) from the client (e.g., 10), and an upper bound on the amount of information to receive from Places data within that radius of the client (e.g., 5). The radius must be at most 50 km, and the information bound must be at most 20 items, since that's all that the Places API supports conveniently. The server responds with a AT message in the same format as before, giving the most recent location reported by the client, along with the server that it talked to and the time the server did the talking. Following the AT message is a JSON-format message, exactly in the same format that Google Places gives for a Nearby Search request (except that any sequence of two or more adjacent newlines is replaced by a single newline and that all trailing newlines are removed), followed by two newlines. Here is an example (with some details omitted and replaced with "..."):

```
AT Riley +0.263 riahi +34.068-118.445 16142
{
 "html_attributions": [],
 "results": [
    {
    "geometry": {
        "location": {
            "lat": 34.06808535000347,
            "lng": -118.4449157000694
        },
        "viewport": {
            "northeast": {
                "lat": 34.0694343302915,
                "lng": -118.4435667197085
            },
            "southwest": {
                "lat": 34.0667363697085,
                "lng": -118.4462646802915
            }
        }
    },
    ...
 ],
 "status": "OK"
}
```

Here I omit the rest of the values in the json, but it can be up to 20 depending on the bound and location. To obtain this information from the Google

Places API I had to create and send an HTTP GET using the aiohttp library. (The API key that I use to retrieve this data has been omitted from my script and needs to be replaced if being used)

2.4 Flooding Algorithm

Servers communicate to each other too, using AT messages. I implemented a simple flooding algorithm to propagate location updates to each other. Servers do not propagate place information to each other, only locations. Servers will continue to operate if their neighboring servers go down, that is, drop a connection and then reopen a connection later. Based on the connection links in figure 1, whenever a server receives a location update from a client, it will propagate that message to each adjacent server. Those servers will then do the same. In order to stop infinite propagation, I had to ensure that a server does not continue to propagate if the time stamp of the message is earlier than the latest received time stamp for that client. That, when a server receives a message it has already propagated, it will end the cycle. Using this type of propagation is really useful since this application is easily able to add/remove servers and still have multiple avenues to fulfill a client's request.

2.5 Implementation

I built my application using Classes, mainly a client and server class. The client class was simple, I just open an asyncio connection to a given socket and send all user inputs through that socket and read for any server outputs.

The server side was more complicated. I also built a server class that contains its name, ip address, port information, logging information, and client information. To start up the server, I use asyncio to start the server until I exit:

```
await asyncio.start_server(func, ip, port)
```

This code will start a server on ip ip an port port and the server will constantly run the func function to handle inputs. To handle inputs I do a simple reader.read call to receive the input from client and then process that input using async functions. Each time any sort of I/O is performed in this application (e.g. waiting for client input, http requests to Google, etc...) the await keyword is used along with an asynchronous function so that the entire process does not get bottle necked due to that I/O.

2.6 Problems Encountered

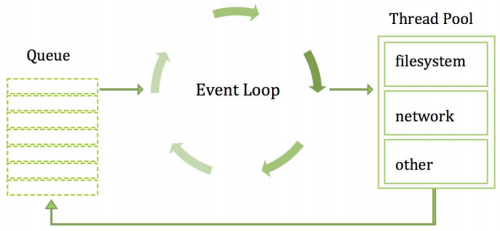
I encountered some difficulties in figuring out when and how to use my async functions. The documentation, however, helped clear things up by giving me a few examples of the 'async' keyword along with the 'await' keyword. I also encountered some issues with my flooding algorithm, since my logic was not flawless and occasionally a server would receive a propogated message but fail to update its client information. Overall, the problems encountered were not too bad and the asyncio library was decently straight forward.

3 Python Asyncio Evaluation

In this section I will go over some details about Python's asyncio library, its overall framework and comparison with Node.js.

3.1 Asyncio Overview

Figure 2: Aysncio Event Loop



Asyncio is a library to write concurrent code using the async/await syntax. Asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and webservers, database connection libraries, distributed task queues, etc. The main motivation for asyncio, as

has been previously stated, is to handle servers with multiple clients at the same time such that the server side is not a bottleneck. Asyncio is most suited for I/O intensive programs because it has great cooperative multitasking library. Slow, I/O intensive tasks can voluntarily take breaks and let other tasks run.

The asyncio library provides two main keywords: 'aync' and 'await'. The 'async' keyword defines a function as a coroutine. This means that this function can suspend its execution and give control to another coroutine. The 'await' keyword suspends the execution of the current coroutine until the awaited function is finished. Whenever a function is awaited, this places that task in the event loop shown in figure 2. Once the awaited function has completed its task (usually some sort of I/O) it waits patiently inside the event loop queue until the the event loop allows the coroutine to start executing again. This provides an efficient way to multitask I/O intensive code such that the slow I/O processes do not bottleneck the entire system.

3.2 Type Checking

Python is a dynamically typed language. This means that Python does all of its type checking at run time, whereas statically typed languages like Java perform the type checking at compile time. This is an advantage for Python because it makes the code simple and easy to use and gives the programmer the benefit of not having to worry about using proper types (especially when working with a new library such as asyncio). One problem with this is that type errors are not caught until run time. This could be a problem if your code base is very large and a small type error is not caught until mid way through execution during a full employment. Java does not have this issue, since once the code is compiled, all type checking has been completed and these errors will not pop up.

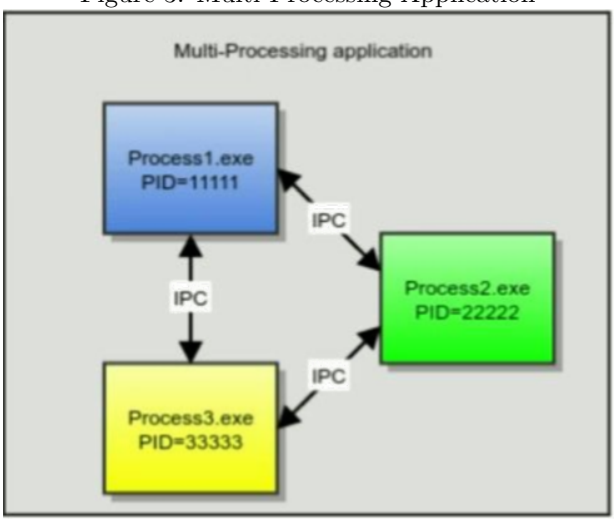
3.3 Memory Management

Python uses a simple form of Garbage Collection to manage its memory. Python uses reference counts in order to free unused memory. That is, each time a reference to an object is created, Python increments the count for that object, and when those pointers are deleted, Python decrements the count. Once a count reaches zero, Python will free that storage. This is a great way to do Garbage Collection and is a lot faster than other approaches (Java); however, there can be problems with circular data structures.

In contrast, Java's Garbage Collection is based off the Java Memory Model which implements a Mark and Sweep algorithm for GC. This means that Java's GC start at the root references and follows all valid pointers to all objects and marks them as in use. Afterwards, Java goes through the Heap and frees all objects that were not marked. This is slower than Python's algorithm, but is safer in the case of circular data structures.

3.4 Multithreading

Figure 3: Multi-Processing Application



Python does not support true multithreading, instead it uses multiprocessing. Multiprocessing is like multithreading in that there are multiple threads with shared memory; however, these threads to not run in parallel. This is because Python has a Global Interpreter Lock (GIL) that prevents multiple threads from executing Python bytecodes at once.

The reason for this is that, since Python memory management depends on reference counting, there could be race conditions with multithreading.

In contrast, Java does use true parallelism, which could allow for more speedy applications. However, for our cases, this is not a huge worry. We can always just add more servers to handle more load and we do not require multithreading in our application since our servers can handle the inputs with plenty speed. When examining my logs, I see AT messages such as this:

```
AT Riley +0.002379417 client ...
```

In this message we can see that the latency between client and server is very minimal (about 0.00237 seconds) which implies that the multiprocessing approach works with enough speed. If we wanted more, we could easily add more servers with asyncio.

3.5 Asyncio vs Node.js

Node.js is an asynchronous event-driven JavaScript runtime designed to build scalable network applications. Many connections can be handle concurrently using the callback method. This method differs from asyncio's await keyword in that, once an event is completed, the callback method is called for that event, as opposed to simply resuming execution. Both methods are fairly straight forward and easy to use, and I would not place one over the other in terms of quality. Javascript is like Python in that it is also an interpreted, dynamically typed language with a single threaded design. Node.js does, however, have the ability to support multithreading on a multiple core environment. From the Node.js documentation: "Child processes can be spawned by using our child_process.fork() API, and are designed to be easy to communicate with".

3.6 Reliance on Python 3.9 Asyncio Features

While dealing with older versions of Python can be a pain for asyncio, it is not too much trouble. Features such as 'asyncio.run' and 'python -m asyncio' are very important, but, as is stated in the documentation, there are work arounds in the asyncio library for older versions of Python.

4 Conclusion

In this paper we have looked into Python's Asyncio library through the implementation of a proxy Google Places API application server herd. We looked into how the server was designed and how asyncio was used to implement the handle of messages from clients concurrently and the task parallelism involved with heavy I/O. We analyzed the asyncio library and compared to Node.js and also evaluated it for this kind of application. Overall, I believe that asyncio is a great tool for building this style of application and that it can be used for a much larger scale program.

5 References

https://docs.python.org/3/library/asyncio.html
https://nodejs.org/en/about/
https://web.cs.ucla.edu/classes/winter21/cs131/hw/pr.html
https://ccle.ucla.edu/pluginfile.php/4220818/mod_resource/content/0/week9.pdf