CS 131 Python Project: Proxy Herd with *asyncio*

David Chen, *UCLA*

Abstract

LAMP is a sufficient architecture for running dynamic web sites and web apps, but it is lacking in certain areas in part due to its maturity and also its components. So, we will seek to analyze asyncio and compare its suitability to LAMP's in creating a proxy herd.

Introduction

LAMP is a workable solution for building dynamic web sites and web apps, such as Wikipedia; however, they have some key weakness. For one, Apache is not suited for services where large amount of data are constantly being transmitted. Additionally, due to the nature of the infrastructure, it lacks scalability once the framework is in place, and also is lacking in mobility. While this system is sufficient for Wikimedia, we run into issues as we try to implement and consider the following:

1. High frequency updates
2. Access from multiple protocols
3. Highly mobile clients

Thus, we need to find a solution that allows for these points by easily; the key points being that servers need to be easily added, accessed and maintained, and that more protocols than HTTP need to be supported.

The architecture that may cover all these points is an "application server herd"; this design involved the servers all communicating directly to each other so that a device can connect to any of the servers in the herd and still be able to access the same data. For this purpose, we will look into Python's asyncio networking library, which should be suitable for rapid data distribution due to its event driven nature. Namely, we will examine its maintainability, reliability, scalability, ease of use, and its interaction with Python's handling of types, memory and multithreading.

1. Asyncio

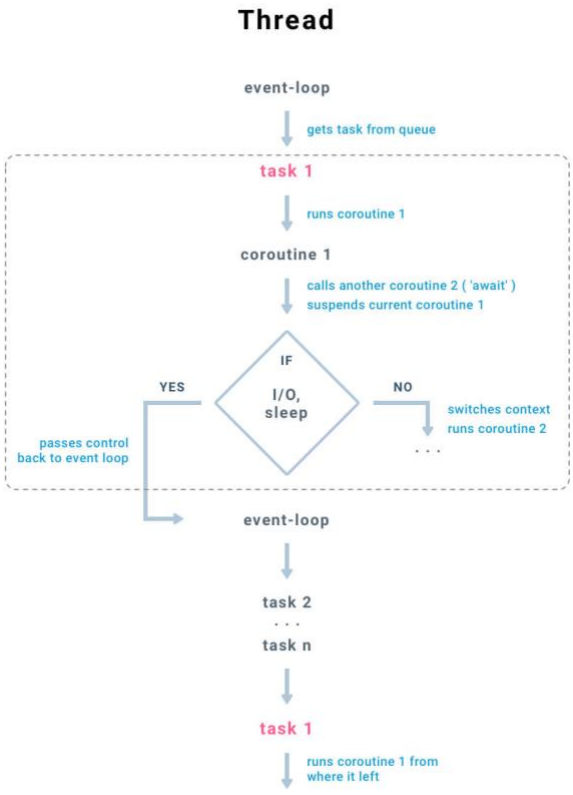

Fig. 1 Single Threaded Event Loop. [1]

1.1. Overview

Asyncio is a python module that allows for single-threaded concurrency in contrast to other methods for parallelization such as python's multithreading module. Its behavior is similar to cooperative multiplexing to schedule events; this works well for network projects such as ours, as it allows for actions to be taken in the waiting time after requests are sent and before the response arrives. It is important to note that this does mean that we have no guarantee on the order in which these events will finish. In this section, we will explore asyncio for Python 3.6.4.

1.2. Event Loop

The asyncio module is based upon the singular event loop that runs on one thread. This is similar to event loops in other frameworks in that it accepts tasks that need to be executed, executes them, and everything in between, from scheduling, delaying, or canceling them. This allows programs to be run more efficiently by making use async/await to run functions in an interwoven manner. Loops can be created, ran and ended with the following functions:

```
loop.get_event_loop()
loop.run_forever()
loop.close()
```

A generic way to think about an event loop would be a while(1) loop with many "if A do B" code blocks within, while asyncio allows you to "pause" a function and return to it later using callbacks.

1.3. Coroutines

Coroutines are what allow the aforementioned "pausing" to happen. The can be made simply with "async def" in contrast to standard "def" function declaration. Coroutines are run within the event loop by creating a Task, a type of Future. This allows the Coroutine to be scheduled, per se. A simple example of a coroutine is as follows:

```
async def a():
    print('a')
    await asyncio.sleep(5)
```

When these Coroutines are run, control is released back to the event loop whenever an "await" is reached; the event loop will then decide the next task to be run. [2] In the above code segment when the Coroutine is run, 'a' would be printed and then control would be given back to the event loop as asyncio.sleep(5) is awaited.

1.4. Futures/Tasks

Futures can be described as an object that holds the result or future result of a Task. These can be made by using the following functions on an awaitable object (such as the code block in 1.3. Coroutines):

```
asyncio.ensure_future(coro, loop)
loop.create_task(coro)
```

These functions can be seen as scheduling Coroutines to run.

By using event loop, coroutines, and futures/task, asyncio achieves a very clean implementation of single threaded concurrency.

2. Prototype Analysis

2.1. Implementation

In the interest of having a scalable design, the herd server was implemented with only one python file, which takes one argument (server_name) and run the corresponding server based on it. The components of the server can be split into three main parts, a ServerClientProtocol, a HerdProtocol and an aiohttp awaitable.

2.1.1. ServerClientProtocol

This is the primary coroutine within the server, made with:

```
coro=loop.create_server(protocol,
host, port)
```

This is responsible for all incoming messages, wither by the client or the other servers in the herd. It accepts 3 types of messages, IAMAT, WHATSAP, and AT. Upon receiving any of these, the server will check if the command is valid and if they are, begin processing the command. If they are not, the server will simply reply back with the string '? ' along with the original message.

IAMAT commands have 3 space-separate arguments, client_id, client_location, and client_time. This analogous to POST on a RESTful framework, in that it updates the information stored on the server. Upon receiving this, the client_location is check using a combination of regex matching and value checking, while time is confirmed simply by seeing if it can be converted into a date time. At this point, the server will compare the received information with the local data. If the information is more recent, the server will update the information and proceed to flood the server herd (described in 2.1.2. HerdProtocol). I decided to store the information in AT message form for so that flooding would be easier. In both cases, the server will then respond to the client with an AT message with the server name, time difference between the server time and client time and a copy of the IAMAT message.

2.1.2. HerdProtocol

This protocol is used to flood the servers and update all the serves with any new information. It takes a lazy approach by simply sending the new information to all neighboring servers in the form of the aforementioned AT message. It does not care if the neighboring servers are down (though it does take not of it) nor does it wait for a response. It then is the receiving server's job to check the information and decide if it needs to continue flooding the servers. The logic is simple – if the information is new, flood the servers, if it's not, there is no need. This implemented by making a client-coroutines using for each neighboring server with:

```
coro=loop.create_connection
     (protocol, host, port)
```

2.1.3. Google Places

On the client side, this is comparable to making a GET request in RESTful server in the form of a WHATSAT command. WHATSAT commands have 3 space-separate arguments, client_id, radius, and bound. Radius is an argument in kilometers for use in the Google Places API and bound is an argument to determine how many results we want. Both are restricted to ranges of 0-50 and 0-20 respectively. After checking the command, we make a http request to Google Places using aiohttp. An awaitable is made with aiohttp function and then scheduled with:

```
asyncio.ensure_future(coro, loop)
```

Here we use await within the aiohttp functions to give control back to the event loop was we wait for Google Places' response.

2.1.4. Logging

All I/O, connections, and disconnections are recorded for each respective server. Below is a sample of a server log during standard runtime (with some truncations):

```
INFO - Serving on ('127.0.0.1', 16675)
```

```
INFO - Connection from ('127.0.0.1', 55686)
INFO - Data received: 'IAMAT'
INFO - Data sent: 'AT'
INFO - Closing client socket: ('127.0.0.1',
       55686)
INFO - New connection with Server: Holiday
INFO - Data sent to Server:(Holiday) AT
INFO - New connection with Server: Wilkes
INFO - Data sent to Server:(Wilkes) AT
INFO - New connection with Server: Hands
INFO - Data sent to Server:(Hands) AT
INFO - Connection from ('127.0.0.1', 55692)
INFO - Data received: 'WHATSAT kiwi.cs.ucla.
       edu 10 5'
INFO - GET: Google Places for kiwi.cs.ucla.edu
INFO - Data received from Google Places
INFO - Data sent to Client:(kiwi.cs.ucla.edu)
       Google Places
INFO - Closing socket: Google Places
INFO - Connection from ('127.0.0.1', 55701)
```

2.2. Analysis

Event loops have always been suited for making network applications such as web sites or servers; this is especially true for server herds, as the flooding will create a fairly significant increase in traffic in comparison to free standing servers. In addition to responding the clients, these servers also have to make requests to Google Places, acting like a client itself.

Asyncio and asynchronous event-driven scheduling is especially suited for this, as there are many gaps in time which a request has been made and the program is waiting for a response. If programs like this were made to be synchronous, there would be much wasted time while waiting for responses. In addition, asynchronous allows for many processes to be run without blocking. For example, a server at any point in time, may have to handle an IAMAT, process flooding from other servers, and be waiting for a response from Google Places at the same time. Being able to run asynchronously improves performance significantly, especially at with high traffic.

In addition, the asyncio module code is much cleaner than what trying to write something from scratch. While this holds true for many of python's libraries, this is especially true for asyncio. The code is incredibly readable, holding true to Python's reputation, and is easy to plug in and use. Asyncio for Python3.5 onward is especially simple as it replaces the previous `@asyncio.coroutine` tag and `yield from` with async/await for coroutines. Protocols also further abstracted network connections and using aiohttp saves the hassle of construction an HTTP request from scratch.

It is very simple to scale this herd server; given a port number and client name, the number of servers can be increased by adding in each server's name, port number and neighbors and running

```
Python3 server.py <Server Name>
```

All servers are running off of the same source code. However, it should be noted that data persistence is an issue in the project. Data will only be maintained as far back as the server has been running. Thus, a client contacting a newly started server will not have access to any of the data from before the server started, even if all other servers in the herd have that data.

2.3. Issues

There weren't many issues in coding the project surprisingly. I was already reasonably familiar with python programming and have experience with asynchronous programming and writing servers from CS111 and various Node.js and Django projects. Asyncio's documentation was already fairly clear about how to use the function once I familiarized myself with

the terminology. However, some challenge was met at expected points, that is, the flooding algorithm ad communicating with Google Places. The issue with flooding places was determining how to prevent infinite loops while still flooding the whole herd. This was easily circumvented, however, once I realized I could take a lazy approach in only caring about starting the flood and leaving it to the recipients to decide what to do after. As for accessing Google Places, the challenge was in incorporating the aiohttp library with asyncio and the proper use of async/await. However, aiohttp provided adequate examples in using the library, so after reading the documentation and quickstart guide, it did not prove much of a challenge either.

3. Further Recommendations and Analysis

3.1. Interaction with Python

Python is an extremely easy to use language, known for having much more readable and simple code in comparison to other languages such as JavaScript or C. It is also very extensive in its supported libraries and abstracts away some of the more complex details of implementing an asynchronous server herd as ours. However, python does come with some drawbacks.

One potential issue is python's use of dynamic typing. This is both good and bad, a double-edged sword of sorts. It allows for quick implementation and makes for rapid development, as one does not have to worry about making sure all types match. However, this also makes for some harder to catch errors where variables are not the expected type. Also, it makes code harder to read as one cannot immediately tell what type a variable is supposed to be. While this is not an issue with short code like ours, determining a code segment's operation becomes exponentially more difficult as the code base increase. In both these case, one may find themselves trawling through function calls in order to under the code.

Python's memory implementation is fairly common in that is uses a heap; this is usually private, managed completely by the interpreter. [3] The exception to this is the memory allocation functions exported from C. However, python's garbage collection is fairly quick, as it keeps reference counts and frees the memory once the object is no longer referenced. This in contrast to other garbage collectors which do not immediately free memory when an object is not referenced. This helps in scalability as we never have to wait around for the garbage collector to free 'unused' memory.

Asyncio uses a single thread, which has certain advantages and disadvantages. Using a single thread minimizes performance's correlation with the machine the server is running on, which makes it nice for something like a server herd, where it is preferable to have similar performance across the herd and to not significantly increase strain on the processor as the size of the heard is increased. However, this makes it harder to scale in terms of processing power as multithreading can take advantage of the increase in the number of processors.

3.2. Compared to Node.js

Asyncio and Node.js actually share a lot of similarities; both are asynchronous, single threaded, and event-driven using an event loop. [4][5] However, notably, Node.js was developed specifically for web applications so holds a few advantages within that scope. [6] It is built on Chrome's V8 JavaScript engine and alone is very lightweight. This makes it fairly simple to deploy something using Node.js.

Another notable difference is the languages they are built upon. While they both use callbacks (JavaScript has promises instead of Futures/Tasks), the modular and simpler nature of python makes development easier and more beginner friendly as inexperienced users of Node.js will often find themselves in the ever so lovingly described ‘callback hell’, in which tracing the flow of the program become increasingly complicated and difficult to understand. However, using Node.js does also give the benefit of having the front-end and back-end share the same language.

While Node.js I currently highly popular and thus causes many applications to be built in it and packages to be made for it, Python still holds an advantage in its maturity having extensive access to libraries for whatever function one may desire. However, ultimately Node.js and Asyncio are suited for different purposes.

4. Conclusion

Python’s asyncio library is extremely well suited for network applications such as a server herd but determining which one one will use when developing is largely depended on what the end product will be used for. If one is creating an asynchronous web app, it may be worthwhile to look into Node.js or various Python alternatives, such as uvloop or Tornado for Python, which boast speed and performance improvements over asyncio.

5. References

[1]https://djangostars.com/blog/asynchronous-programming-in-python-asyncio/

[2]https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/

[3]https://rushter.com/blog/python-garbage-collector/

[4]https://eng.paxos.com/python-3s-killer-feature-asyncio

[5]https://hackernoon.com/asyncio-for-the-working-python-developer-5c468e6e2e8e

[6]https://www.netguru.co/blog/use-node-js-backend