Application Server Herd Design

Ankith Uppunda, *COM SCI 131*

Abstract

I have been tasked into looking into asynchronous programming for designing an application server herd, where they communicate directly to each other. We look to see if asyncio network library can replace LAMP, since LAMP has bottlenecks with high traffic, mobile clients and access on other protocols than HTTP . To test this out, I wrote a service herd in asyncio that is a parallelizable proxy for the Google Places API.

1. Introduction

To explore asynchronous programming, I implemented a server herd consisting of 5 servers (Goloman, Holiday, Hands, Wilkes, Welsh). Each server talks bidirectionally to the servers that it is allowed to talk to. To make sure that the information reached all servers, I implemented the flooding algorithm to make sure each server received the data. A client can either declare where it is with IAMAT or it can query from the Google places API what is around there with WHATSAT. Whichever server is queried will send back to the client the information that it requested using asyncio.

2. Research on Asyncio

To understand how to implement my server herd, I started off researching asynchronous programming and specifically the library asyncio. To find this information, I looked documentation, example code, and articles.

2.1 Event Loops

Event loops are an integral part of asynchronous programming. Event loops are task managers that can register, execute, and cancel calls. They can also launch subprocesses to communicate with an outside program. These are the basis of our server, since they execute coroutines(See 2.2).

To create an event loop with asyncio:

```
loop = asyncio.get_event_loop()
```

Events can be scheduled with:

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

These functions schedule a coroutine to execute in an event loop. They create a future object for the loop to evaluate at another point.

To run loop use:

```
loop.run_forever()
```

and to close the loop use:

```
loop.close()
```

These functions were the backbone of my project, as my project had an event loop that would schedule a function whenever the client sends my server a string.

2.2 Coroutines

Event loops schedule coroutines to execute. Coroutines are an integral part of the asyncio library. Coroutines are components that generalize subroutines for multitasking by allowing multiple entry points for suspending and resuming execution. Event loops schedule these coroutines using the code in the previous section.

An example coroutine would be :

```
async def client_handle(reader, writer):
    writer.write(await reader.readline())
```

This is a really simple snippet of code which basically writes back whatever was sent to it. The async keyword makes this function a coroutine. When an event loop creates the task using create_task(), it does not run the function, it just creates a task to execute that function later in the future. When the event loop decides to run its tasks, client_handle will be called. The await keyword extracts the actual return value of the coroutine. Await creates a suspension point that tells the event loop that an I/O operation is about to take place which allows it to switch to another task.

To run client handle:

```
loop = asyncio.get_event_loop()
c = asyncio.start_server(
      server.client_read, ‘127.0.0.1',
       port, loop=loop)
server = loop.run_until_complete(c)
```

This code was taken straight out of my proxy herd project. In context of the project, this code starts the server at the specified port and host, with a specific coroutine which was server.client_read in this case.

This returns a future object, which is basically a coroutine that is still yet to run. Finally,
I pass the future object into the loop.run_until_complete method to actually run the coroutine. This is simpler to how I implemented it, but this is the main idea.

These two ideas Event loops and Coroutines was the main idea in my prototype.

2.3 Asynchronous
Event loops are asynchronous not synchronous, which means that the order that the coroutines are run in are not known ahead of time. This means that your code wait on a coroutine, since it will interleave the execution of coroutines. If done right, it will not get blocked by a request, as it will continue executing the code after the request without waiting for the request to complete. This is a very powerful tool, as it allows a environment similar to multithreading without the hassles of actually multithreading.

Event loops are not multithreaded; when I first started this I thought I could keep making event loops and it would create its own thread. I thought that get_event_loop() would give you a new event loop each time. However, this is wrong; event loops are single threaded, and unless you explicitly create more threads it will stay that way. Event loops are asynchronous instead running on one thread. Coroutines are interleaved together, and the event loop will not wait on the output of a coroutine request. Instead it will move on and execute the next piece of code.

Since the loop is asynchronous, it would not have to wait for the output of GET request to the Google places api. Without the asynchronous property, the code would stall at the request waiting for it to finish. It is a good thing that a server can move around and continue its other processes without worrying about what the request produces.

3. Design of my Herd
In this section, I go over my design of my prototype. This provides background for the next section. The majority of my implementation is located in the *serverhelper.py*, which contains a Server class and a Client class. My *config.py* class contains constants that are used throughout my code.

On the higher level, the server class contains the data for the server, the name of the server, and the port of the server. The data of the server is the previous requests from clients. It also contains the handle method that handles IAMAT and WHATSAT calls.

The client class is a protocol class that allows a server to create a connection to another server. This allows it to connect and send data to the other server without stopping the current coroutine client_read. This implementation is here because I did not design my server class to use protocols. Looking back, this would have been a more elegant solution to the problem.

3.1 Using Streams instead of Transport
For the overall server design, I ended up using streams instead of transport. Using the start_server method in the asyncio, I create a server that runs client_read whenever a client sends something to the server. I used reader and writer streams to get the information from the client and send it back to the client.

3.2 Client Read

My client read method had four parts. The first part read the line from the client using the *reader.readline()* method.

Once it got the command it split of into three parts depending on the type of command it was. If it was a server then it ran the "AT" command; if it was a IMAT then it ran the IMAT command; if it was a WHATSAT then it ran the WHATSAT Command.

First the command is checked to be valid by the method isValid, checking if the input is valid. If it isn't, my program responds with : *? command*, depending on what the command is. If it is valid, it checks what the keyword is.

A valid IMAT command is of this form:

```
IMAT Client LAT-LONG Posix-time
```

If it was a IMAT command, then my program strips the latitude and longitude, time, and subtracts the perceived time from the time put in. Using this, it sends back a command of the form:

```
AT Goloman +860990.9653711319
 kiwi.cs.ucla.edu +34.068930-118.445127
 1520023934.918963997
```

It sends an AT with the name of the server that the client sent the IMAT to; the first number is the time difference from server to the clients time, and the next thing is its latitude and longitude. The last thing is the posix time from the IMAT command. After the AT, it sends the AT to the other servers that it talks to.

A valid WHATSAT command looks like this:

```
WHATSAT Client Radius Amount
```

For a WHATSAT command, my program did a little more. It first checks if the server has the client that the WHATSAT command contains. If it doesn't then the server sends an error to the client using the writer stream. If not, the server loop will send out a GET request to Google Places API to get the json information that it needs. Once it gets the data it will relay it back to the client side in the manner asked in the spec. I used aiohttp to create a Http get request. I used a class called ClientSession to make a session tied to my event loop. Using that session, I made a GET request with the parameters that Google needs. This is all asynchronous, as the event loop will not wait for the get request to come back.

The last part is the AT handling. This part handles the flooding algorithm. It sends the AT string with the servers visited concatenated to the end of the string. This makes sure that it doesn't end up in an infinite loop. One server might get the same AT call twice, but it will eventually end. Using protocols, I created a connection to the servers that I needed to send the AT string to.

The code looked like this:

```
transport, protocol = await
   loop.create_connection(Client,'127.0.0.1', port)
```

This creates a transport and protocol based off of my Client class which inherited protocol. The transport writes to the connection specified by the host and the port.

My logs show if the servers that it sends the information too is online or offline.

Example Logs:

```
DEBUG:root:Running command IAMAT
kiwi.cs.ucla.edu +34.068930-118.445127
1520023934.918963997

DEBUG:root:Server Hands not set up yet
DEBUG:root:Server Holiday not set up yet
DEBUG:root:Server Wilkes not set up yet
DEBUG:root:Command Produced: AT Goloman
+859109.7708489895 kiwi.cs.ucla.edu
+34.068930-118.445127 1520023934.918963997
```

The logs show that the other servers that Goloman talk to are not up, so they will not get the AT response. This is the log for one IAMAT response.

3.2 Problems while working

I had a lot of problems implementing the GET request in the beginning mainly due to the fact that I didn't really understand asyncio and aiohttp. The first error that I got was an SSL error, but that was fixed with adding my own connector. The second error I got was the program ended up stalling at the request. This was due to having my code in a separate method and I fixed it by putting it back in the main method.

Another problem that I had was with the flooding algorithm having infinite loops. I fixed that with appending the server that it came from to the message.

I had a lot of problems with the run_forever() method. Since I needed to add more coroutines to the event loop in order to talk to other servers, I first used open connection. Since this wasn't working, I looked at the docs and saw the transport class, and looked at an outside implementation

4. Suitability

My research has led me to the conclusion that asyncio is suitable for the problem of creating an application server herd.

4.1 Asynchronous benefits

If there is a lot of traffic as in there are a lot of requests, asynchronous is the best match since it will not hang on the requests, but instead move through the code asynchronously. This is much better than the normal LAMP method, which will hang on every request. The servers are always wait for connections from client, and since its asynchronous connections will not clash with each other.

If the design was synchronous, it would have to wait for a connection to finish before it goes to another connection.

The WHATSAT command really shows this benefit, since a synchronous server would have to wait for it to return before evaluating another command. This could really stall the program, which illuminates why asynchronous is the better choice.

4.2 Asyncio Benefits

Writing an asynchronous server from scratch would be hard and filled with bugs. Asyncio's library makes this a lot easier, abstracting a lot of the methods for making asynchronous code which makes this a lot easier to write.

They provide things like 'open_connection', 'create_connection', and transports and protocols. Event loops abstract a lot of the difficulties since they basically schedule the coroutines for you. This made the code much easier to write.

Transports are also very useful, as asyncio allows them to be used easily. They can be written to, read from, and closed whenever the programmer pleases.

Protocols allow the programmer to specify how he wants the data to be received or sent. These abstractions allow the user to have a very easy time coding versus coding the connections, and asynchronous capability on their own.

The library creates the connection to the host and port that you specify, which makes this a lot easier to implement. They create the server for you with the 'create_server' function.

4.3 Python Benefits

I think python is the best language to be writing this in. Mainly since it scales well on other machines. This method is not dependent on the spec of the machine its running on. Due to the fact that asyncio is single threaded, it will scale well to other machines. Whereas the Java-based approach will probably be multithreaded which largely will depend on the amount of cores that a user has on its system if they want to run a server herd. It will slow down significantly for systems with less cores.

Another reason python is the best language for this is string and data manipulation is very easy in python. Most of the checking can be done with the built in string and list functions, whereas in C++ or Java there would probably be a lot of helper functions to parse the data thats coming in.

Also the amount of helper libraries make processing data really easy. For example, the JSON package made it really easy to take in Json and make a data structure out of that data.

5. Possible Concerns

There are concerns with asyncio. Namely there are concerns in regards to type checking, memory management, and multithreading.

5.1 Type Checking

Python doesn't use static type checking unlike other popular languages. Instead, python uses dynamic type checking which entails that types are only checked at runtime. This can cause runtime errors since it would only find a mismatch at runtime. This doesn't make it more unreliable though. As long as testing is done reliably and carefully, python is a great language for creating a reliable server. It doesn't have to be more unreliable than say C++ or Java.

The biggest pro of dynamic checking is the development time. It is much easier to develop with dynamic checking, since the programmer doesn't get these type errors that are really annoying in Java and C++. While more runtime errors occur, there is more time to debug, since putting out code is much easier. There is less starting time, since I don't have to explicitly think of types and can code how I please. With an application like this, there would probably be a lot of runtime problems with every implementation however. Due to the nature of multi-threading, it would be really hard to write a bug free server for C++ or Java. Python is a great solution because of that, because while it has problems with the runtime errors, all the languages would have a lot of runtime problems in the development stages.

Another problem with Python's type checking is it is very hard to read a python program. Due to the fact that there are no explicit type definition, the reader is often left confused. Java and C++ are easier to read and deduce what the method is doing. If this is a big group coding this project, then a lot of the coders will be left confused. A way to fix this problem is to document. Documentation for this case is very important since it is very hard to figure out what the programmer intended in some cases.

5.2 Memory Management

Both java and python use heap for allocating data, but there garbage collectors(GC) operate in different ways. Java's GC destroys objects when they are no longer in use at an unknown time. Python uses the reference count method, which enables it to immediately delete objects that are not used.

For the server herd, Java's GC would be faster since it wouldn't delete until necessary. If there is not a lot of memory usage with the servers, then java's GC would be better, while if memory is scarce, then python's method will be better for getting memory quickly for the server herd. With a small server herd, either method shouldn't be a problem, but java's will be a bit better. However for a large server herd, python will definitely scale better, since it will get rid of places of memory not in use much faster. I would recommend Python because it scales for much larger systems, but if the size is around the prototype, then I might suggest Java.

5.3 Multithreading

Multithreading is probably why we wouldn't implement the herd with python. While asynchronous works great for everyone, multithreading will scale better the more powerful the computer is. If we are developing these servers to run on computers that run on a lot of cores, multithreading will be the way to go, because it will do everything at the same time instead of jumping around like in an asynchronous program. As the number of cores increase, we get closer to processing everything at the same time as it comes into a server, which will always be better than the asynchronous solution.

Python would be the solution if this is a product we are scaling for commercial use. Since most people don't have access to top of the line computers, they would not get the benefits of multithreading, and we would definitely use python in this case.

Basically, the solution is situational, as in both are good for different things.

6. Node.js vs asyncio

Both methods are asynchronous and single-threaded used for networking. They both support multi-threaded applications.

One of the main problems of Node.js vs python is that Node.js is very new. There are less developers in the market, whereas python is very mature, and has a lot of library support. Python is also extremely portable. Another problem is readability for python applications; as the code grows it is very hard to read due to its typing, whereas Node.js is easier to read.

Overall python's asyncio and Node.js are very similar, as Node.js use promises which are the analogous to the callback feature on asyncio. Overall Python is probably better because of the amount of support and portability that it provides over Node.js.

7. Conclusion

I recommend asyncio over the other solutions. It is very easy to create big programs, as they abstract a lot of the difficulties. There are a lot of pros for using asyncio and not a lot of cons. The cons are pretty specific, so for general purpose, I would recommend asyncio. Asyncio works very similar to Node.js, so they are definitely both worth checking out.

8. 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*, docs.python.org/3/library/asyncio.html.

[2] Cannon, Brett. "How the Heck Does Async/Await Work in Python 3.5?" *Tall, Snarky Canadian*, Tall, Snarky Canadian, 17 Dec. 2016, snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/.

[3] Foundation, Node.js. "About." *Node.js*, nodejs.org/en/about/.

[4] Seha, Uroosa. "Node.JS vs Python: Which Is Best Option for Your Startup." *Vizteck*, 21 Jan. 2018, vizteck.com/blog/node-js-vs-python-best-option-startup/.