Python Asyncio Library for Networked Server Herds

Alexander Ma

University of California, Los Angeles

1 Introduction

A server herd is a system of servers that passes messages to each other along with their clients. The messages the servers pass to each other ensure that the servers have the same data, so a client can communicate with any server, and receive the same response. The Python asyncio module can be used to create such a system.

The asyncio module was introduced in Python 3.4 [10]. This module provides an API to run coroutines, perform network communication, control subprocesses, and synchronize concurrent processes [5]. The use of these API's in creating a server herd are discussed below.

2 Asyncio Features Used in the Server Herd

Coroutines in Python can be created by adding the async keyword before the definition of a function. Then, these functions can be executed using one of two methods:

asyncio.run(coroutine) This method was introduced in Python 3.8, and it runs the coroutine, taking care of the event loop setup and management [10]. However, it is recommended that this method only be called once per program.

await coroutine(args) This method provides a blocking wait on the coroutine. The current code does not continue until the coroutine completes. If this is used within an event loop, however, other coroutines can be executed during this waiting time.

Event Loop Manipulation Using the event loop directly allows greater control over the execution of the coroutines in loops, but may be less reliable and require more effort to create.

I decided to use event loop manipulation to start the server, because Python 3.8 is the newest version of Python, so using asyncio.run(coroutine) may not be a reliable strategy if older versions of Python are used, and there are no running coroutines before the server starts. Creating and running event loops manually was not too difficult. My code to implement a similar functionality is as follows:

```
def run_until_interrupted(self):
      loop = asyncio.get_event_loop()
      coro =
asyncio.start_server(self.messageParser
, self.ip, self.port, loop=loop)
      server =
loop.run_until_complete(coro)
      try:
            loop.run_forever()
      except KeyboardInterrupt:
            pass
      # Close the server
      server.close()
loop.run_until_complete(server.wait_clo
sed())
      loop.close()
```

The above code is quite short, with a similar structure, so the lack of the asyncio.run(coroutine) method is not too much of an impedance for the purposes of creating a server herd [8].

Within the messageParser method, there are multiple times await is used, since those methods occur during the execution of a coroutine. Due to the way coroutines are executed, these awaits allow other coroutines to run at the same time [6]. For example, the following log from my Singleton server demonstrates this:

```
New connection to ('127.0.0.1', 42282)
New connection to ('127.0.0.1', 42284)
SERVER1 AT Campbell
+0.0006053447723388672 Hello
+34.068930-118.445127
1583112934.3206089
Attempting to connect to 12201
SERVER2 AT Campbell
+0.0006053447723388672 Hello
+34.068930-118.445127
1583112934.3206089
Connection closed with ('127.0.0.1',
42284)
Connected to 12201 . Writing data.
Closing connection to 12201
Connection to 12201 closed
Attempting to connect to 12202
Connected to 12202 . Writing data.
Closing connection to 12202
Connection to 12202 closed
```

```
Attempting to connect to 12203
Connected to 12203 . Writing data.
Closing connection to 12203
Connection to 12203 closed
Connection closed with ('127.0.0.1',
42282)
```

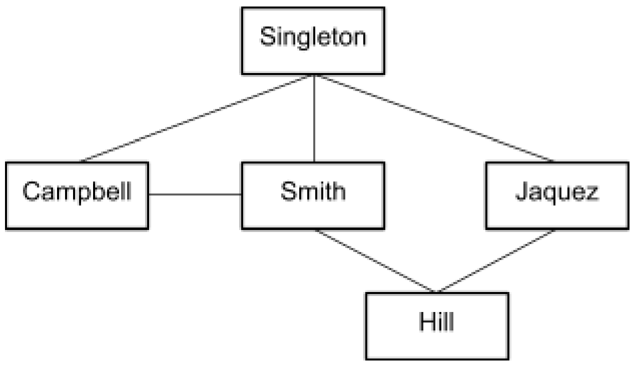
The log shows that the server handled two messages from two different clients at the same time, despite the server being run on a single thread. First, a new connection to a client at port 42282 is connected. After the connection is created, the connection helper function uses await while waiting for a message to come in. While the coroutine is waiting for the message, a second coroutine can run and handle the new connection to port 42284. Then, the second coroutine waits for a message using await. The first coroutine receives the first message then uses await while waiting to connect to Jaquez (which is at port 12201). While waiting, the second coroutine receives a message and processes it. Since the second coroutine receives an end message, it closes the connection without await. Then, the first coroutine can run uninterrupted to its end, since there are no other coroutines. This strategy works well when there is little computation between awaits, but may decrease in efficiency if the computation between awaits increases.

To create the servers, I used asyncio.start_server, and to create the clients, I used asyncio.open_connection. Each of these methods is a coroutine, so other coroutines can run while waiting for these methods to start [9]. asyncio.start_server has a coroutine as one of its arguments, which should accept a StreamReader object and a StreamWriter object as arguments, in that order. These objects are used to read from and write to the client. asyncio.open_connection returns a StreamReader object and a StreamWriter object, which do the same thing, except asyncio.open_connection reads from the server and writes to the client. The read operations are coroutines, because messages arrive at unknown times. The write operations are not coroutines, because the time of the message being sent is known (the time of the write operation).

3 Server and Server Herd Design

The server receives two types of messages from clients. The IAMAT message gives the server the location of the client with the client's name. The IAMAT message is received through the StreamReader, and there is no response from the server. The WHATSAT message is a request from the server for locations near a client. This message is also received through the StreamReader. The information of nearby places are not on the server, but on the Google Places API on Google Cloud. Therefore, when a server receives a WHATSAT message, the server sends an http request through the aiohttp library, which uses coroutines, to Google Cloud [1]. Then, the server parses the Google Cloud response, and sends the parsed response to the client through the StreamWriter.

The server herd consists of 5 servers, which are connected bidirectionally as shown below:



Each of the servers is not directly connected to at least one of the other servers, but each of the servers has at least one path to each of the other servers. Therefore, to pass information from one server to all other servers, some of the other servers have to pass the information again to the servers that have not received the information.

To complicate the information passing further, one or more of the servers may be down at any time. Therefore, all paths to be tried in case one of the paths is down. The maximum length of a path is three, so the number of times a message needs to be passed is three. To implement this message passing, I created three interserver messages: SERVER0, SERVER1, and SERVER2. SERVER0 is for the first server that sends out the information, SERVER1 is for the second server that sends out the information, and SERVER2 is for the third server that sends out the information. These three messages prevents infinite message passing by setting the limit to the number of messages in the message definitions, but also guarantees that the message will get passed to all servers, if all the servers are interconnected.

Since servers can go down and up at unknown times, the server implementation needs to be able to handle other servers going down and up. My solution to this potential problem is that the server connections are only up when information is being passed. This way, there is no need to check if a server is still up, or to check if a server has come up, because the creation of the connection already checks if the server is up. This method of connection should not slow down the server too much, since these methods are coroutines, and thus other processes can run while waiting to connect.

4 Performance of Asyncio

As mentioned earlier, asyncio runs using coroutines, which means that while one coroutine is waiting asynchronously, any other coroutines can run. This increases the efficiency of the server over a purely sequential implementation. Coroutines are also slower than multithreading, though coroutines should not be slower by much, because the majority of the time in network communication is waiting for messages, which is not lost time with the coroutine implementation. Therefore, coroutines are not slower by much with small server loads, though this may become a greater issue with large server loads. If performance is an issue, multithreading can be simulated by running multiple servers to distribute the load, like in a server herd.

However, creating a server herd adds to the complexity of the program. Programs in Python may encounter runtime errors, because there is no type checking before the program is run. The more complex a program is, the higher the chance that one of these type errors is encountered. Therefore, complex servers implemented using asyncio may be unreliable. This unreliability is offset by greater flexibility, however.

Creating a server herd also requires more memory. Each server in the herd requires a copy of the same information, which creates redundancy. This means there is a much greater amount of memory used by a server herd compared to a single multithreaded server. On the other hand, the memory management is much more reliable, because only one thread changes the same memory at the same time, so there are no issues where two threads attempt to change the same memory location.

5 Comparison to Java

There are other languages with which the server can be implemented. Java has networking and multithreading libraries, so Java can be used to create a server herd [3,4]. Java uses multithreading, which can be faster than coroutines, but multithreading creates race conditions, so synchronization measures have to be implemented. Synchronization causes a decrease in speed, so synchronized multithreading may be slower than coroutines if there are many synchronized memory changes.

The runtime errors from type checking should not be an issue in Java, because Java has the option to use static type checking, so type errors should be caught before the server is run. Furthermore, types are more visible in Java, since they are given with each variable declaration, so type errors are also easier to fix. This means Java servers are probably more reliable than Python servers.

A single multi-threaded program can be used in Java instead of multiple server programs in a server herd. This means less memory is used, since there can be less redundant memory. However, this causes the program to use synchronization, which may slow the program, as mentioned before. Java has an inbuilt keyword synchronized, which makes it easier to synchronize programs in Java compared to Python.

6 Comparison to Node.js

Node.js is a JavaScript library intended for use in network applications, such as websites [2]. Like Python asyncio, Node.js uses event loops and coroutines to run its networking. The Node.js implementation of type checking is also similar to Python, with runtime type checks.

Unlike Python, however, Node.js controls the running of the event loop instead of the user, and the entire program runs in the event loop [2]. This means the configuration of the event loop is more reliable, and programmers can concern themselves more with the other parts of the application. Furthermore, since Node.js and its language JavaScript are intended for use in web applications, Node.js many of the web functionality builtin. This makes it more convenient to use Node.js for applications such as this server herd that communicate with external web servers, which mostly run on JavaScript.

One example of this convenience is JSON (JavaScript Object Notation) object parsing. In JavaScript, JSON objects are parsed directly from JSON strings, but Python turns JSON strings into a Python dictionary, which is not as simple [7]. This makes Python programs less efficient than JavaScript programs at JSON object parsing. Since many web applications use JSON objects, web servers and clients that use Python are less efficient than Node.js servers in this aspect. The time lost from JSON parsing may become significant if JSON object traffic is frequent or if the JSON objects are large.

Since Node.js is more closely designed to the intended purposes of the server herd, while still having an overall architecture similar to Python's asyncio module, Node.js seems to be the better framework to use in this server herd, though Python's asyncio module can still be used to create a similar effect, albeit with more effort.

7 Conclusion

Using the Python asyncio module to implement a server herd seems feasible. The test servers in the server herd I created were able to successfully pass messages to the clients, external web servers, and to each other. The server herd was able to handle dropped servers and clients, and have good time performance through the use of coroutines. Furthermore, memory management was not an issue, since the coroutines run one at a time in an event loop, switching between each other whenever the running coroutine encounters an await, thus avoiding race conditions in memory accesses. However, coroutines in an event loop can be slower than multi-threaded programs if the time between awaits in the coroutine is large compared to the time waiting for the awaited statement to complete. Furthermore, dynamic type checking in Python provides flexibility, but at the cost of run-time reliability.

Java and Node.js are two possible alternatives to Python. A Java implementation could use multi-threading with synchronized memory changes to implement a server herd. Although static type checking would decrease the chance of run-time crashes, a reliance on synchronization may make memory management tricky and less reliable. Node.js on the other hand, behaves similarly to Python. Node.js also uses coroutines and dynamic type checking, with the corresponding benefits and drawbacks. However, Node.js has some advantages in web applications, which make it more suited than Python for the purposes of creating a server herd that uses http requests.

References

[1] Aiohttp contributors. Welcome to AIOHTTP. 2018. https://aiohttp.readthedocs.io/en/stable/

[2] OpenJS Foundation. *About Node.js*. https://nodejs.org/en/about/

[3] Oracle. *Lesson: Concurrency*. 2020. https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

[4] Oracle. *Package java.net*. 2020. https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html

[5] Python Software Foundation. *asyncio — Asynchronous I/O*. 2020. https://docs.python.org/3/library/asyncio.html

[6] Python Software Foundation. *Coroutines and Tasks*. 2020. https://docs.python.org/3/library/asyncio-task.html#coroutine

[7] Python Software Foundation. json. 2020. https://docs.python.org/3/library/json.html

[8] Python Software Foundation. Python asyncio library. 2020. https://github.com/python/cpython/tree/master/Lib/asyncio

[9] Python Software Foundation. *Streams*. 2020. https://docs.python.org/3/library/asyncio-stream.html#asyncio-streams

[10] Python Software Foundation. *What's New in Python*. 2020. https://docs.python.org/3/whatsnew/index.html