CS 131 Project Report
Assessing Python's asyncio Library for an Application Server Herd

Abstract

Server herd architectures are useful for inter-server communications involving rapidly-evolving data. In this report we assess the suitability of Python's asyncio asynchronous networking library as a tool for implementing an application server herd. As an exercise, we use asyncio to create a parallelizable proxy for the Google Places API. Afterward, we briefly compare the approach of asyncio to that of Node.js and the use of Python to a Java-based approach. Finally we evaluate whether the use of Python and asyncio is suitable for this use case.

1 Background

1.1 What is a Server Herd?

A server herd consists of multiple servers across which information is synchronized through direct communication between servers. This is a departure from many traditional application architectures, in which peripheral web servers generally receive information from a central application server. Many websites implement some form of this design; Wikipedia and all of its related sites rely on a central Wikimedia application server. However, in situations that involve frequent updates and require fast response times, the central server introduces a bottleneck, giving way to the server herd architecture.

Say we have three application servers 1, 2, 3, and 4. The application servers don't all have to "talk "to each other; rather, each server just needs to be connected to every other server by some path, and all communication is bidirectional (in other words, the communication between servers can be represented by a connected undirected graph).

For instance, say we have the following relationship: 1 talks to 2 and 3, and 2 and 3 talk to 4. If a client application to sends some information to any of the servers, then the others will receive this information after one or two communication cycles between servers, all without having to talk to some central database.

Finally, an important feature of the server herd architecture is that it is persistent even in the event that one server crashes. When this server is back up, it should continue to receive messages as before.

1.2 About asyncio

Asynchronous I/O refers to the ability of a program to perform some sort of processing before input/output operations are fully completed. This is important since much of the time, input/output operations are a performance bottleneck in that they are both slow and unpredictable.

Thus, asynchronous I/O provides a type of concurrency, but it is a single-threaded, single-process design: it uses what is known as cooperative multitasking. [2] This concept will be examined in more detail throughout the report.

In Python, we can implement this asynchronous I/O functionality using asyncio. From its official documentation:

> *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 web-servers, database connection libraries, distributed task queues, etc.
>
> *asyncio* is often a perfect fit for IO-bound and high-level structured network code.
>
> *asyncio* provides a set of high-level APIs to:
>
> - run Python coroutines concurrently and have full control over their execution;
> - perform network IO and IPC;
> - control subprocesses;
> - distribute tasks via queues;
> - synchronize concurrent code [1]

1.2.1 Coroutines and Async/Await

While asyncio relies on concepts such as callbacks, events, transports, protocols, and futures, the basis of the package is coroutine objects and the async/await keywords.

A coroutine is like a function that can suspend its execution before reaching control and indirectly pass control to another coroutine for some amount of time. We define a coroutine using the *async* keyword. Inside a coroutine, we can use the *await* keyword to pass control back to the *event loop*, which coordinates the overall execution of the asynchronous block of code. [2]

For illustrative purposes, we have here an example of a simple coroutine.

```
async def g():
   r = await f()
   return r
```

In this coroutine $g()$, the *await* keyword tells the event loop to stop the execution of g until $f()$ has returned. In the meantime, the event loop can run another block of code which is ready for execution. We can only use the *await* keyword with "awaitable" objects, which are coroutines, tasks, and futures.

Coroutines can be wrapped into "tasks" with *asyncio.create_task*(). Tasks allow us to schedule coroutines concurrently. Here, $f()$ is a future, which represents the eventual result of an asynchronous operation. While we can explicitly do many lower level operations with tasks and future, they are mostly unnecessary for our prototype. [1] [2]

In previous versions of asyncio, we had to use a relatively convoluted interface to manage the event loop. However, in more recent versions such as the one we are using, we can make a simple call to asyncio.run().

2 Our Google Places Proxy

It would be disingenuous to evaluate asyncio without actually using it in the context of some project. At the same time, creating some large project for the sole purpose of using Python and asyncio would likely be more trouble than it's worth. Thus, for the purposes of this evaluation, we've created a lightweight proxy for the Google Places API to simulate a server herd.

2.1 Behavior

The servers in our herd accept TCP connections, in our case taken care of by asyncio, from clients as well as each other. These TCP messages, which are meant to emulate mobile device users with location information, can be of a few different forms. Any message that does not follow one of the formats described below is treated as an invalid command, to which the server replies with a '?' and a copy of the invalid command.

2.1.1 IAMAT

As the name implies, IAMAT messages allow clients to tell the server where they are. The format is as follows:

```
IAMAT kiwi.cs.ucla.edu +34.06-118.445127
1520023934.918963997
```

where IAMAT is the message type, "kiwi.cs.ucla.edu" is the client ID, "+34...-118..." is the location in latitude and longitude, and 152.., is the POSIX time of the message according to the client.

2.1.2 WHATSAT

Again, as the name implies, WHATSAT messages allow the clients to send a query about places where other clients are located. The format is as follows:

```
WHATSAT kiwi.cs.ucla.edu 10 5
```

where WHATSAT is the message type, kiwi.cs.ucla.edu is the name of *some other* client, 10 is the query radius in km from that client. and 5 is the limit on the amount of information to receive. This information comes from the Google Places API.

2.1.3 AT

The server responds to clients with an AT message that looks like:

```
AT Hill +0.263873386
kiwi.cs.ucla.edu +34.068930-118.445127
1520023934.918963997
```

where AT is the message type, Hill corresponds to the server ID that received the message, and +0.26... is the time difference between the server receiving the message and the client sending it. The remaining fields are from the previous IAMAT message. If the server is responding to a WHATSAT message, the format is the same, but with a JSON-format message following the AT response.

AT messages are also used for servers to send information to each other. Whenever a server receives a message from the client, it also provides the AT response to all of the servers via a *flooding algorithm*. This flooding algorithm dictates that upon the generation or receipt of an AT message, each server propagates, or "floods" the message to all of the servers with which it was designated to communicate. Since all of the servers are connected, the message eventually reaches all of the servers.

2.2 Servers

Our prototype consists of five servers with IDs "Hill", "Jaquez", "Smith'", "Campbell", and "Singleton" that communicate amongst themselves with the following pattern:

- Hill talks with Jaquez and Smith.
- Smith talks with Campbell.
- Singleton talks with everyone but Hill.

We can represent this communication graph with the following Python dictionary, which is used in our code:

```
server_connections = {
'Hill': ["Jaquez", "Smith"],
'Jaquez': ["Hill", "Singleton"],
'Smith' : ["Hill", "Campbell", "Singleton"],
'Campbell' : ["Singleton", "Smith"],
'Singleton' : ["Jaquez", "Campbell", "Smith"]
}
```

2.3 Usage

We create a server instance by simply running

```
python3 server.py <server_name>
```

3 Implementation

Here we briefly describe the implementation of the server herd in Python with asyncio.

3.1 General Functionality

Below is our top level *asyncio.run*() call as described in the Background section:

```
try:
   asyncio.run(server.run_forever())
except KeyboardInterrupt:
   pass
```

The coroutine that we pass into *asyncio.run*() is the *run_forever*() method of our Server class, adapted from the TA hint code. This coroutine, shown below, uses *asyncio.start_server*() and the corresponding *asyncio* server object's *serve_forever*()method.

```
async def run_forever(self):
server = await
    asyncio.start_server(
     self.handle_echo, self.ip, self.port)

   # Serve requests until Ctrl+C is pressed
   async with server:
   await server.serve_forever()

   # Close the server
   server.close()
```

We pass the *handle_echo*() function, also adapted from the TA hint code, into *asyncio.start_server*. In *handle_echo*, we handle the client messages and do all the necessary processing.

3.2 HTTP Requests Using aiohttp

Since *asyncio* doesn't inherently support HTTP requests, we had to also use the *aiohttp* library, which supports asynchronous HTTP requests. The Google Places API call using *aiohttp* is shown below:

```
async with aiohttp.ClientSession(
       connector=aiohttp.TCPConnector(
            ssl=False,
       ),
) as session:
async with session.get(url) as resp:
response_dict = await resp.json()
```

3.3 Flooding Algorithm

Finally, we have our flooding algorithm, which is actually quite simply. The *server_connections* dictionary, which is shown above, gives the servers with which a server directly communicates. We simply iterate through that list of servers, sending the correct message to each.

In order to do so, we use *asyncio.open_connection*(), which returns a pair of $(StreamReader, StreamWriter)$ objects [1] that have a number of built-in methods we can use for our TCP connection. This makes sending and receiving data extremely easy, since we don't have to use callbacks or low-level protocols and transports.

```
for server in server_connections[self.name]:
  reader, writer = await
    asyncio.open_connection(
       '127.0.0.1', server_ports[server])
  writer.write(message.encode())
  await writer.drain()
  writer.close()
```

Since each server is running its own instance of the *server.py* module, the message will propagate as expected.

4 Initial Assessment of asyncio

4.1 Advantages

One major advantage of *asyncio* was the ease of use; **it was very easy to write an asyncio-based program to run and exploit a server herd.** While this was partially due to the fact that we were coding in Python (more on this later), *asyncio* certainly provides a very usable interface to asynchronously handle I/O operations, specially now that managing the event

loop is much easier using *asyncio.run*(). Almost every lower-level detail associated with the prototype was already implemented in some way in *asyncio*, and we just used some convenient, high-level interfaces like *asyncio.start_server*() and *asyncio.open_connection*().

While the inability to make HTTP requests using only the asyncio library might initially be interpreted as a disadvantage, its a testament to *asyncio* that the *aiohttp* library was available to allow us accomplish this very easily.

No library will ever cover every use case, but ideally they can be integrated for any necessary purpose. This was an initial concern of *asyncio*, since most common libraries won't necessarily be compatible with the asynchronous paradigm by default. For example, we weren't able to use the popular *requests* library for our API calls. However, the fact that we were able to integrate *aiohttp* instead was a very good sign, and in fact, there are actually a number of non-blocking libraries work with async/await. They include *aiojobs* for managing background tasks, *async_lru* for an LRU cache, *aiofiles* for file I/O, and *umongo*, a MongoDB client, among numerous others. [2]

In addition to the specific benefits of the *asyncio* library, the overall choice to asynchronously handle I/O was very effective. This allowed each server to perform other operations while listening for inputs and waiting for HTTP responses instead of blocking the rest of the code from being executed. This *event-driven* paradigm enabled updates to be processed and forwarded rapidly throughout the herd. Timestamps from the log data generated from testing the server herd supports this conclusion.

4.2 Disadvantages and Problems

One problem that came up during the creation of the prototype was the difficult nature of debugging *asyncio* code. Asynchronous code is not necessarily executed sequentially, so understanding erroneous behavior of the server, even with proper logging capabilities, was cumbersome.

A disadvantage of the asynchronous implementation as opposed to a purely synchronous implementation is that it's much more likely the bugs are generated from the unordered processing of client messages. For example, if a client sends an IAMAT update and then sends a WHATSAT query corresponding to the same client ID very shortly afterward, there is a possibility that the WHATSAT message is completely processed before the IAMAT update, leading to unexpected behavior.

Meanwhile, a purely synchronous implementation would execute strictly sequentially, meaning this problem wouldn't exist. However, we'd have a fairly significant performance bottleneck that would likely outweigh this potential for bugs.

In order to really complete our evaluation of *asyncio*, we must also compare it to the popular alternative Node.js as well as address any concerns about Python compared to a Java-based approach.

5 asyncio vs Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, often used for building fast and scalable network applications. It uses an event-driven, non-blocking I/O model that is extremely similar to the one described above for *asyncio*.

5.1 Programming Interfaces

Both Python and JavaScript are dynamic programing languages in which parallelism is not well supported (more on Python's Global Interpreter Lock later). However, both provide similar support for *asynchronous concurrency*, using the *async* and *await* keywords, callbacks, and promises/awaitables. [3]

One difference between is that in *asyncio*, asynchronous functions will never be executed until and unless another asynchronous function awaits it, whereas in Node.js, code inside an async function will always be executed up until the await statement. In addition, with *asyncio* we have to manage the event loop in order to properly execute asynchronous code, while this extra mechanism is not required in Node.js: asynchronous code is handled properly by default. [4]

At this point, it may seem that Node.js provides an even simpler interface for asynchronous programming. However, the lower-level API's provided by asyncio to create and manage event loops, implement protocols using *transports*, and integrate async/await code with lower-level code involving callbacks (recall the background discussion of futures). In many cases, such as networking and handling OS signals, the nuance provided by these low-level API's is probably preferable to that of Node. [3]

5.2 Performance

In terms of performance, Node.js likely has a fairly significant edge on *asyncio* (although the *uvloop* package advertises that it can greatly speed up asyncio code). Both approaches remove the I/O performance bottleneck, so the overall performance difference may not be that large depending on the application. In general, if high performance is really desired, an application would likely benefit from the use of Node.js, but *asyncio* certainly doesn't have any glaring performance concerns. [3]

5.3 Development Logistics

Another important factor to consider is potential integration into an existing project. Node.js code is written in JavaScript, which is also very prevalent for front-end application programming. This provides developers with the unique opportunity

of having an entire web application written in a single language, and this might prevent some integration issues that could arise otherwise.

However, there is also an advantage to using Python. Python is a much more general-purpose language, whereas JavaScript is used almost exclusively for web and network applications. In a large project, it's very likely that some tasks (such as data analysis or the low-level operations) can be accomplished only in Python, or at least much more easily. Thus, for larger applications, asyncio might greatly simplify the development process.

While the developer community for Node.js is very large, asyncio is gaining popularity as well, and both receive fairly frequent updates. In fact, *asyncio* is fairly new and received a pretty large overhaul in Python 3.7 and 3.8. For this project, we based all of our research on Python 3.8.2 which added the *asyncio.run*() functionality and the *asyncio* REPL.

While the *asyncio.run*() functionality is convenient and the REPL may be useful in some cases, **developers can probably get by with slightly older versions of Python**.

6 Python vs Java Concerns

6.1 Type Checking

Python is dynamically typed, meaning that that all type checking is performed at runtime rather than compile time. Thus, type errors are encountered at runtime rather than compile time. Python's type inference employs a concept colloquially known as "duck typing," which refers to the fact that any object may be used in any context until it is used in a way that it doesn't support (for example, a method is called that the object class does not provide).

One downside of dynamic typing may result in slower resolution of bugs than in statically typed languages, as bugs are only discovered when the program executes the buggy portion of code. An advantage, however, is that dynamically typed languages such as Python do not require type declarations, whereas many (but not all) statically typed languages such as Java do. The lack of type declarations typically makes code much shorter and simpler.

Overall, **dynamic typing and Python are pretty widely accepted and should not pose any sort of problem to the development of any application**. If static type checking really must be employed, there are libraries that implement static type checking in Python.

6.2 Memory Management

Both Python and Java offer automatic memory management, meaning that programmers don't have to manually allocate and deallocate certain blocks of memory. The main difference between the Python memory manager and the JVM's memory management is in garbage collection.

Java's generational garbage collector employs the mark and sweep algorithm. In short, Java places objects into generations. At certain time intervals, the generations are updated, at which point certain objects are "marked" and the remaining are "swept."

Meanwhile, Python employs automatic garbage collection using reference counting. As the name implies, Python simply counts the number of references to an object remaining in the code to keep track of its objects. There are some bugs in this approach, such as circular object references. A benefit of this is that memory is freed promptly in many cases, which may be useful in a server herd. One major disadvantage is that Python's garbage collection isn't compatible with multithreaded applications, resulting in the Global Interpreter Lock (GIL), but in terms of actual memory management, there isn't any major issue with Python. [5]

6.3 Multithreading

Python does not support multithreading, while Java does. Some alternatives to multithreading in Python are multiprocessing using the standard *multiprocessing* library, which generally doesn't provide too large of a performance boost, and writing a Cython package to implement multithreading, which is labor-intensive. **If multithreading is a required characteristic of the application, then we certainly shouldn't use Python. However, in our case, the performance bottleneck of the server herd is likely addressed adequately addressed by asynchronous concurrency.**

7 Conclusion

Overall, the experience using *asyncio* for the Google Places Proxy was very positive. Further research and comparisons revealed some pitfalls, but for our case of building a more comprehensive, server-level application, *asyncio* **is suitable.**

8 References

[1] *asyncio - Asynchronous I/O*
https://docs.python.org/3/library/asyncio.html
[2] *Async I/O in Python*
https://realpython.com/async-io-python
[3] *Python vs Nodejs, Which is Better for Your Project?*
https://da-14.com/blog/python-vs-nodejs-which-better-your-project
[4] *Intro to Async Concurrency in Python and Node js*
https://medium.com/@interfacer/intro-to-async-concurrency-in-python-and-node-js-69315b1e3e36
[5] *Basics of Memory Management in Python*
https://stackabuse.com/basics-of-memory-management-in-python