Project: Proxy Herd with Asyncio

Shaan Mathur, *UCLA Computer Science Undergraduate*

Abstract

Asyncio is a Python module that allows asynchronous, event-driven programming. We evaluate its application in an application server herd with a network of communicating servers, and find that its lightweight nature and Python's succinctness both make using Python as a good choice for creating a server. We also address Node.js and evaluate how both languages can be used to gain the best of both languages.

1. Designing an Asyncio Server

1.1 What is Asyncio?

Asyncio is a Python module designed specifically for event driven applications. More specifically, it is a single threaded library meant for achieving concurrency without having to deal with a large portion of the synchronization issues true parallelism brings about. At a high level, the module has an event loop that one can schedule coroutines with.

Event based systems are a particularly powerful paradigm to apply in server side programming. Servers must be able to quickly respond to any new or existing client connections, thereby achieving low latency. Servers must also be able to serve many clients at once, thereby achieving high throughput. Writing code in a non-event driven manner can hurt both of these metrics; if a server is too busy performing a synchronous I/O operation in order to serve one client, it is wasting time not serving other requests (low throughput) and thus causing overall response time to increase (high latency).

The use of Asyncio in this project is to examine whether this module makes Python a suitable language for development of a simple server herd topology.

1.2 Server Design with Asyncio

Because Python is an object oriented language, a natural approach to beginning our design is to take an object oriented approach. At an elementary level, we have some sort of Server object, which has a unique identifier and peer servers to communicate with, as well as being able to accept TCP connections and serve requests.

1.2.1 Protocols

Asyncio offers various ways to form TCP connections, the main two being either via protocols or streams. With protocols, we have an abstraction that falls strongly in line with the Server abstraction, since protocols essentially provides an interface for Servers to implement. This is extremely convenient in terms of development, because now all one has to do is provide Asyncio some sort of instance of Server that implements various (and simple) protocols such as 'connection_made' or 'data_received'. Servers can thus just focus on handling requests rather than trying to properly configure any TCP/IP specific settings. In that sense, protocols are a strong development tool.

However, if the server wants to initiate a TCP connection itself, doing so is a bit awkward with this abstraction, since it requires designing a protocol class that the Server class itself uses to initiate connections. Though this is not terrible since it reinforces object oriented programming, Asyncio offers another mechanism that we use in this report's prototype: streams.

1.2.2 Streams

Another abstraction that Asyncio offers us are streams, which are essentially wrappers for the protocol abstraction. To use streams, we simply invoke 'open_connection' specifying with whom we would like to connect with, and we are returned a reader, writer pair. Both of these elements can essentially be thought of as byte streams: the code can read bytes from the reader using 'reader.read()' or write bytes over the connection using 'writer.write()'.

One huge advantage of this approach is that we can now open connections arbitrarily in the code without much difficulty; this is better than the other option for protocols because now we can remove some of the verbosity of designing an entire protocol.

Another large advantage is that reads and writes can be done asynchronously, whereas protocols only were invoked on receipt of messages without any potential asynchronous calls. This is what makes this abstraction more powerful than the former, because now we can achieve a larger degree of concurrency than we would with just protocols.

The only caveat of streams is that this abstraction is not as elegant as simply requiring our Server to inherit from the Asyncio.Protocol. In our application, this con is

outweighed by the pros, due to increased concurrency and the ability for the Server to make arbitrary TCP connections with Google and other Servers in the herd with ease.

2. Building Servers with Python

Clearly Asyncio is a viable module to build event-driven asynchronous code in Python, but how does Python stand as a tool to build a full fledged server or herd of servers?

2.1 Dynamic Type Checking

One of Python's key features is that it is dynamically typed; though this is a great feature to have when wanting to program quickly, it can be problematic for writing industrial quality code. The main issue is that with dynamically tested code, bugs may not be revealed until runtime. Though this is acceptable when debugging programs during development, one does not want an invalid type exception to be unhandled in the field, crashing an entire server. This is why Java seems like a strong language to use here: it is statically typed, and so most exceptions having to do with incorrect types have been analyzed at compile time. Further, though one might argue that Python makes its code more debuggable by being able to quickly read error messages and find the line where the misstep occurs, Java's debug symbols can be effective enough that an error message can indicate what went wrong and where.

This concern, though a valid one, is largely mitigated by the succinctness and brevity of actual Python code. A lot of code in Python can actually be reduced to a few lines, sometimes even into a "one-liner". For instance, when submitting an API request to Google Places, one requires a comma separated longitude, latitude pair in the URL; the input provided, however, may not have this comma separated. Very quickly, one can write the following line to get the job done:

```
‘,’.join(re.findall(r’[+-][0-9]+\.[0-9]+’,
msg)
```

Python code is very brief (our functioning server took only 163 lines to implement, whitespace included). Because of this, high code coverage is not too difficult to achieve, our modules are small and therefore maintainable, and we have code that is simple (and a pleasure) to read.

With respect to performance, Just In Time compilers for Python exist (e.g. pypy), and so any runtime performance checks will become negligible with such extra tools at hand.

2.2 Memory Management

Memory management in Python uses a garbage collector, and achieves garbage collection by using a combination of reference counting and (when resources are nearly exhausted) some form of mark-and-sweep. Other higher level languages such as Java use other garbage collection methods that cause the application to block for an extended period of time, though it occurs not so often. Nevertheless, Java's garbage collection, if not invoked by the code itself at strategic points in the code, will essentially be a blocking, synchronous operation that requires the entire program to halt; Python on the other hand will very rarely do this since it uses reference counts as a primary mechanism. Python thus wins in this sense, though Java can have its garbage collector invoked at times of low client demand or if one server can temporarily go down for quick maintenance/resource freeing.

Of course, this does not compare to effectively written C++ code that self manages its resources. However achieving perfect resource management gets more difficult the larger a project becomes, and C++ is certainly more verbose than Python. Therefore as the scale increases, it is more likely that C++ code will have memory leaks as more programmers work together.

2.3 Multithreading

Here we have chosen to take an asynchronous event-driven programming approach to creating instances of the Server class. Another conceivable approach would be to have multiple threads who would have the sole job of reading/writing/handling requests, etc. In this manner, a multi-core machine would achieve much more efficient use of CPU time, which could be very valuable in terms of performance. Python has support for multithreading as well, but Java has a wide array of synchronization primitives that would make it seem fairly advantageous over Python in this respect.

The issue with taking a multithreading approach is however the need for so many synchronization primitives. With single threading, a large number of race conditions are mitigated and control is much more streamlined. Though true parallelism is sacrificed, the code becomes much more maintainable and extendible since one does not need to worry about whether locking primitives are being used correctly. So although Java would be effective for multithreading with synchronization support, the idea is that we want to avoid these complications in the first place.

3. Asynchronous Programming - Node.js

A fairly large elephant in the room in this report is the server side programming language that is a derivative

of Javascript: Node.js. Node.js is an event-driven, asynchronous programming language, designed especially for server code! So instead of using a specific module in Python in order to achieve asynchrony, why not use a language designed specifically for it?

3.1 Object Orientation

Node.js, like Javascript, features prototype object orientation (though ECMAScript 6 now has classes as syntactic sugar). Prototype object oriented programming has its advantages in particular use cases, particularly when we want inheritance to be performed dynamically rather than statically. Though something like this can be achieved in Python by using 'setattr', it is more "native" in Node.js.

More importantly, Node.js itself runs on an event loop under the hood, meaning it is an event driven system inherently. Events handlers are registered to handle incoming connections, which means heavy use of anonymous functions in the code. Other than that, there is not much code pollution and the syntax is relatively light since it is dynamically typed. So Node.js seems well suited for server-side programming.

However, Node.js has some drawbacks that Python still wins with. Most importantly, it is difficult to write synchronous code with Node.js. Nearly everything occurs via events in Node, which is great for most purposes in server side applications, but it lacks much extensibility if we need another back end that takes really requires intensive computation time. This leads to another major point: Node.js does not support multithreading, which means true parallelization is not attainable with Node.js.

Node.js is very strongly designed for server side programming, though, and it would seem a waste to not use it. There are actual npm modules that can link Python code and Node.js code which means committing to one language will not impede later extension.

3. Conclusion

Clearly, Python seems to be an excellent choice for building this server. It's object oriented nature fits nicely with our design choices, and the language itself is succinct enough to keep our code short and thus maintainable. Though Python has various quirks such as dynamic type checking and reference counted memory management, these quirks actually make Python a strong choice.