CS 131 Project. Proxy herd with `asyncio`

Nikola Samardzic
University of California, Los Angeles

Abstract

This report is part of my submission for Project of prof. Paul Eggert's Computer Science 131 – Programming Languages class for Fall 2017.

In this report I analyze if Python's `asyncio` library can be used to build a good framework for developing proxy server herds for large throughput, ephemeral content. The library is compared to other established frameworks such as Node.js and Java multithreading. This report will also include a summary of how different language envirionments effect the quality of our framework as well as the speed and ease of the development cycle. Most broadly I attempt to answer the question: Is asyncio a suitable library for a start-up to get a reliable, effective server up and running fast?

1 Introduction Overview

In comparing a Python proxy server implementation to one written in Java, the main questions that need to be answered are:

1. How does Python's duck typing compare to Java's static type checking?
2. How does Python's Global Interpreter Lock (GIL) imposed event-based approach compare to Java's multithreading capabilities?
3. How do different Garbage Collection implementations effect proxy server performance?

Maybe even more important, due to similarity of the underlying framework, is the question of how `asyncio` compares to Node.js.

2 Multi-threading vs. Event Loop

The GIL is not necessarily a disadvantages, and is praised by many members of the Python community. Possible Python implementations that forgoe the GIL will probably not be integrated into the standard any time soon, partly due to the very demanding criteria imposed by Guido van Rossum (Python's BDFL) including that the implementation must not impose any overhead on non-parallel Python code. [4] So – the GIL from a start-up lifecycle point of view – is here to stay. For the specific needs of our application this is not a big issue, since it is deep in I/O bound territory. All the proxy server code does is:

1. Read and process incoming requests
2. Possibly do a look-up in the internal database
3. Possibly send request to main server
4. Process response from main server
5. Create and forward appropriate data to other servers
6. Create and forward response to client

The only part of this cycle that we have to wait for on the CPU is the internal database lookup, string processing of incoming requests, and creation of response strings. All other actions can be done asynchronously as long as we have adequate support of notification of completion and callback methods (which are of course provided by `asyncio`).

Compared to Java's multithreading approach, the event-based system is easier to reason about and, again, doesn't impose any overheads. However, if our server herd were to suddenly require significant computation to be performed to service requests, we would probably have to completely recode the whole server in a language that doesn't impose a GIL (like Java).

3 Comparison to Node.js

Compared to Node.js (which is also event-based and single-threaded) first note that both `asyncio` and

Node.js are built on top of `libuv`. Node.js is further built on top of V8. V8 allows for compilation of JavaScript code to native machine code. This allows Node.js to partialy forgoe the overhead of interpreted languages. Analogously, `asyncio` can be run through uvloop (implemented in Cython) to forgoe overheads of interpretation.

Benchmarking results on HTTP servers between Node.js and Python's asyncio show noticably better performance in favor of Node.js. However, it turns out that Python is mostly bottlenecked by the HTTP parser in aiohttp (used because asyncio does not support HTTP itself – a notable drawback). [3] propose an alternative to aiohttp called httptools and demonstrate that benchmarks show better performance than Node.js.

4 Ease of Development and Type Checking

As long as the development cycle is concerned, `asyncio` might be very suitable for start-ups and others that might expect very tight edit-compile-test-debug loops to be an indispencible advantage. Namely, due to duck typing, new code can be dynamically loaded into the interpreter. Servers can easily be updated without shutdown (or compilation for that matter). Updates can be easily plugged, tested, and unplugged with ease and without any noticable interruption for the user.

A loose type checking policy also allows us to easily mock methods during code testing. This would be an issue in Java. Specifically for our application there is a big benefit to mocking and prototyping due to two reasons:

1. The client requests must be reliable and easily replicable. Without mocking this would be practically impossible, since we would have to rely on networks, which will not be able to provide the required replicability of test cases.

2. During load testing, we would require thousands of requests to be sent out to Google's API. Aside from this being very slow and hard to replicate, it would also be very expensive. In effect, load testing our servers would also load test Google's servers. We can easily forgoe this issue by mocking the request to Google.

5 Memory Management and Garbage Collection

In terms of memory management and Garbage Collection (GC), Python may pose some challanges for server implementations.[2] Python implements garbage collection through reference counting. This may be problematic for servers, as they are usually intended to run for long times and may easily accumulate memory leaks due to the fact that reference counting will not be able to recognize cycling references as garbage. Of course, Python has a special approach for dealing with these cycles. It has a special place in memory that references all container objects in the current process (note that non-container objects like strings, ints, floats, etc. cannot run into cyclical references because they cannot reference other objects). Every so often Python will look at all container objects currently in memory, and detect all cycles deleting the useless containers. This process is pretty expensive, which is why python does it a lot less often than the regular garbage collection it does through reference counting. Since many connections are being created and lost during the running of the server, there will be many container objects which may easily accumulate cyclical references. This might overload the part of the GC which finds and resolves these references. What's even worst is that due to Python's GIL, any time the GC is running, the whole server is blocked.

Java has a much more elegant solution to garbage collection (especially with regard to servers). [1] Java creates a deamon GC thread whenever a process runs. This thread independently takes care of Garbage Collection. If we have multiple cores on our server, the deamon thread can operate virtually independently of the server's main event loop. This is a big win in worst case performance for Java.

6 Integration into other Frameworks

Python's asyncio is compatible with Twisted and other Asynchronous I/O implementations that came prior to `asyncio` (like callbacks, deferreds, stackless Python, coroutines). In the backend CPython implementation, coroutines share the majority of their structure with generators. Since Cython's uvloop supports asyncio, we can easily integrate asyncio into servers that use compiled, low-level languages.

7 Other Concerns

Python has relatively little packaged out-of-the-box solutions which might speed up the development cycle. This is in stark contrast to PHP, which contains off-the-shelf implementations of discussion boards, char servers, etc. Other languages worth mentioning are Erlang, Scala, and Go. Go has amazing performance, but requires very careful and though-through development, which might impact the development cycle. Erlang and Scala work very well at scale, and Erlang is based on very nice and useful inter-process communication. It is also widely used in the telecommunications industry. Scala runs on

top of the JVM, which comes with its benefits and drawbacks...

8 Conclusion

`asyncio` provides many useful synchornization tools, as well as specific tools that are very useful for building servers (e.g. Transports and Protocols).

Lack of multi-threading does not seem to be of any concern for our specific application since it is I/O bound. The fact that Python is duck typed gives many advantages of prototyping and helps tighten the development cycle in comparison to Java. When benchmarked on the Echo server, it has been shown to be twice as fast as Node.js. [3]

It is also worth mentioning that `asyncio` is easily integrated into lower-level compiled code due to the fact that it is supported by the uvloop project. It also can interact with Java through Jython. Garbage Collection may impose a noticable overhead to Java implementations due to lack of ability to detect and remove cyclicaly reference garbage fast. Generally, Java provides a much more mature API for controlling different parameters of the Garbage Collection process as well as hinting at the GC what should be done (this is supported in Python too, but, again, Java offers more control). Specifically, all of the objects associated with interserver communication have relatively short lifespans, but probably not short enough to be collected by the Young generation garbage collection process. In general GC implementations having to deal with objects that are not ephemeral, but not long-lasting either can impose huge overheads. [1] Java provides us with an API through which these parameters could be tunned and consequences of widespread medium lifespan objects mitigated.

Note also that `asyncio` has many tools that support for debugging. [5]

References

[1] *Java Garbage Collector Documentation*. Oracle.

[2] Neil Schemenauer, *Python Garbage Collection*. 2000.

[3] Sahand Saba, *Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js*. 2014.

[4] *Global Interpreter Lock*. Python Documentation, 2017.

[5] *Debugging with asyncio*. PyMOTW-3, 2016.