A Proxy Server Herd Application With asyncio

Nathan Smith – University of California, Los Angeles

Abstract

With the internet more accessible than ever, a plethora of server designs have emerged the meet the ever-changing demands of people using web applications. This paper examines a possible "server herd" design using Python 3 and Python's asyncio asynchronous networking library for a Wikimedia-style news site, in which frequent updates are expected via mobile clients and various protocols. A demo implementation of a server herd architecture is used to discuss the pros and cons of Python and asyncio, and comparisons to both Java and Node.js are included.

1. Introduction

Wikipedia uses the widely-used LAMP (GNU/**L**inux, **A**pache, **M**ySQL, **P**HP) stack on multiple redundant web servers with a load-balancer. This has obviously proved to be a very successful architecture for Wikipedia: it is the 5th most popular websites on the internet as of March 2018 [1].

The rapid growth of hardware in the forms of mobile devices, tablets, laptops, and routers, as well as software in the forms of new web frameworks and server applications have led to a much greater range of possibilities for web sites and applications. However, with these new technologies and possibilities also come a new learning curve as well—modern server architectures are much more complex than a simple LAMP stack.

This paper looks at a different possible architecture—an application server herd—for a supposed new Wikimedia-esque news site. This site expects frequent updates, access via various protocols, and mobile clients. In order to better accommodate these demands, this paper discusses a demo server herd that has been implemented in Python 3 with the asyncio module.

1.1. Asyncio

Asyncio is a fairly recent addition to the Python standard library, being first introduced in Python 3.4 by PEP 3156 [2]. Since then, both Python 3.5 and 3.6 have seen substantial updates, both feature-wise and syntactically to how the library works. Asyncio aims to give a standard solution to the problem of asynchronous I/O in Python, which had previously been done via solutions with tools such as WSGI, multiprocessing, gevent, Tornado, or Twisted [3]. Asyncio draws a lot from Tornado and Twisted, two popular asynchronous networking libraries that handle asynchronous I/O via selector loops, as well as other asynchronous languages such as JavaScript.

2. Implementation

The prototype application written implements an application server herd architecture, in which a "herd" of servers communicate with each other to act as the application's database. A servers peers are predefined, and servers communicate bidirectionally with each other via a very basic flooding algorithm. Each server can accept three possible commands, and will output an error if an invalid command is given. Each command is described briefly as follows.

The "IAMAT" command is a command sent via TCP from a client, giving a server its location. The server must then give the client a confirmation of receipt, store the client's location, and propagate this client and its location to all other servers.

The "WHATSAT" command is a command sent via TCP by the client to a server. It asks for a Google Places Nearby Search query for the location of a specified client. Parameters are also given to define the radius and number of results.

The "AT" command is a command sent from server to another server, in order to propagate a client's location. A server will keep a record of this client, similar to if it received an "IAMAT" command directly from that client and then send "AT" commands to all of its peers.

Each server also logs any relevant events that occur.

Note that there are improvements that could be made to this demo as it is currently implemented. The flooding algorithm could be replaced with a more efficient algorithm. Another limitation is that if one server in the herd was to go down then come back up, that server would miss any new clients added to the herd and have no way to obtain those clients. Both of these improvements are beyond the scope of this demo, but are important to consider should a application server herd be deployed in a production environment.

For a greater, in-depth explanation of the exact demo application behavior, I recommend you visit the project specification by Paul Eggert at http://web.cs.ucla.edu/classes/winter18/cs131/hw/pr.html.

3. Considerations

If considering Python and asyncio for a project, I would recommend that the following points be weighed and considered.

3.1. Structure

Asyncio is very unopinionated; it gives no standard structure or format that programmers should follow. To many, this is an advantage as it means that it can be better customized and adapted to suit individual needs but it also means that an organized and understandable structure must be architected should asyncio be chosen to be used in any large, production project.

3.2. Future Changes

It seems that Asyncio is still being actively developed and changed. Both Python 3.5 and 3.6 introduced new features including a new async/await syntax, asynchronous generators, and asynchronous comprehensions [4] [5] [6]. It is reasonable to assume the Asyncio's API may take a few more years to stabilize, so ensuring that all asynchronous code is up-to-date may be a non-trivial task.

3.3. Typing

As a language, Python is notable for being dynamically typed. Dynamic typing means that each object is bound to a type at runtime, as opposed to static typing in which each object's type is known at compile time.

Dynamic typing has both pros and cons. Dynamic typing allows programmers to build applications quicker as they don't have to specify types for each variable, but being forced to specify types can help the compiler to catch potential errors before runtime. Since ideally a server herd would be used in production, a small tradeoff in development time for more reliable and less error-prone code seems like a worthwhile investment. Therefore, the dynamic typing in our case is more a con than a pro.

Python 3.6 saw the introduction of variable type annotations which when used in conjunction with tooling can provide much of the safety that static typing does [7]. While Python is still a dynamically typed language, I would strongly advocate the usage of type annotations for any large project as it can mitigate some of the concerns with the type system. Comprehensive testing of code (which would hopefully be done for production applications) can also mitigate type error concerns.

3.4. Asynchronous I/O and Multithreading

Python has support for multithreading through the threading module in the standard library [8]. When using asyncio, multithreading is not required, but is possible. In the demo, I used a separate thread to run a second event loop to fetch data from Google, to see how multithreading with asyncio would function but the same functionality could have easily been implemented in the same loop.

As opposed to handling asynchronous I/O with separate threads or processes, asyncio uses an event loop. An event loop works by making notifying an "event provider" when a particular event occurs. This provider can then call an event handler that executes appropriate code for the event.

Asynchronous I/O is a must for scalability, as it means that new threads don't have to be created as the application gains more users. This means that the single thread loop approach of asyncio has a big advantage over multithreaded applications.

In my experience with asyncio, this advantage comes with a tradeoff of a learning curve: writing asynchronous code is complicated, especially coming from a background of only synchronous code. The library comes with a large litany of terminology and ideas and gives little in the way of examples. Furthermore, many tutorials are already out of date given the syntax changes in Python 3.5 and 3.6. Armin Ronacher perhaps put it best when he wrote: "I can't help but get the impression that it will take quite a few more years for it to become a particularly enjoyable and stable development experience" [9]. I'm inclined to agree. I see the async module in Python as a very good step forward, but think that as of now, the ecosystem needs a little longer to grow and stabilize before writing code with it can become enjoyable. That being said, asyncio does make writing asynchronous code much easier than languages such as Java.

3.5. Memory Management

Python is notable in handling memory automatically with a garbage collector; meaning that programmers don't have to worry about new and free and pointers like in other languages. The Python memory manager uses a private heap to allocate and deallocate all objects and data types [10]. How Python's garbage collector works is dependent on the Python implementation. CPython, the most popular implementation of Python, uses a combination of reference counting and mark-and-sweep garbage collection. This combination of garbage collection techniques is specifically to address the problem of reference counting handling circular references [11]. CPython's implementation of garbage collection is a major pro for our server herd uses, as it means objects will be deleted quickly when they are no longer used in addition to the fact that programmers do not have to worry about manual memory management.

4. Alternative Technologies

4.1. Java

Java is a very popular language that can also used to write networking applications.

Unlike Python, it is statically typed, which provides a greater degree of potential type errors. In this area, Java is better suited for the described server herd needs than Python.

Java is also notable for automatically managing memory with a garbage collector, like Python. Java differs from Python in its garbage collector's implementation. Java's garbage collector is much more traditional. It allocates objects in the heap, and garbage collects when the heap is full. It also divides the heap into two generations, the nursery and old space to optimize memory usage of the heap. Garbage collection works via a mark and sweep strategy, in which all objects that are reachable are marked as alive, and all other parts of the heap are put on the free list [12]. Java's memory management is not as good of a choice than Python's for our server herd. In Python, objects are deleted immediately via reference counting. In Java, objects are not deleted until they're garbage collected, which doesn't have to happen immediately.

Java does not have an obvious equivalent to an asyncio event loop, which means that asynchronous I/O would need to be done via multithreading. For the server herd application, this would mean more threads would need to be created as more clients use the application. A Java implementation would require much more resources as the application grows as opposed to the Python one; the Java approach with multithreading would simply not scale as well. This comparison really highlights the beauty of the event loop, and shows why it is so popular for networking applications. The event loop is probably the biggest reason for choosing Python over Java for a server herd application; any language that does not support an event loop will not be as scalable.

4.2. Node.js

Ever since its initial release in 2009, Node.js has taken the web development community by storm and is Stack Overflow's "Most Popular Framework Technology of 2017" [13]. Unlike Python, which has an asynchronous event loop added on that has to be started by a blocking call, Node.js is a fully asynchronous JavaScript runtime that is asynchronous from the start [14].

Node.js appears to be a much more mature choice for asynchronous network than Python and asyncio. It boasts an multiple amazing package managers and the largest collection of open source packages in the world.

5. Conclusion

Overall, it seems Python and asyncio is a fine choice for a server herd architecture. Python and asyncio's event loop approach and reference counting offers major scalability advantages over languages like Java. However, Node.js also looks to warrant further consideration as its large ecosystem, maturity, and promise of an asynchronous runtime are quite promising.

References

[1] *wikipedia.org Traffic Statistics*, https://www.alexa.com/siteinfo/wikipedia.org.

[2] van Rossum, Guido. "PEP 3156 — Asynchronous IO Support Rebooted: the "asyncio" Module", https://www.python.org/dev/peps/pep-3156/.

[3] Brown, Amber. "The Report of Twisted's Death Or: Why Twisted and Tornado Are Relevant in the Asyncio Age", *Pycon 2016*, http://lucumr.pocoo.org/2016/10/30/i-dont-understand-asyncio/.

[4] Selivanov, Yury. "PEP 492 — Coroutines with async and await syntax", https://www.python.org/dev/peps/pep-0492/.

[5] Selivanov, Yury. "PEP 525 — Asynchronous Generators", https://www.python.org/dev/peps/pep-0525/.

[6] Selivanov, Yury. "PEP 530 — Asynchronous Comprehensions", https://www.python.org/dev/peps/pep-0530/.

[7] "typing — Support for type hints", *The Python Standard Library,* https://docs.python.org/3/library/typing.html.

[8] "typing — Support for type hints", *The Python Standard Library,*https://docs.python.org/3.6/library/threading.html

[9] Ronacher, Armin. "I don't understand Python's Asyncio", http://lucumr.pocoo.org/2016/10/30/i-dont-understand-asyncio/.

[10] "Memory Management", *Python/C API Reference Manual*, https://docs.python.org/3/c-api/memory.html.

[11] “Why does python use both reference counting and mark-and-sweep for gc?”, https://stackoverflow.com/questions/9062209/why-does-python-use-both-reference-counting-and-mark-and-sweep-for-gc.

[12] “Understanding Memory Management”, *Oracle Diagnostics Guide,* https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html.

[13] *Stack Overflow Developer Survey Results 2017,* https://insights.stackoverflow.com/survey/2017#technology-frameworks-libraries-and-other-technologies.

[14] “About Node.js”, https://nodejs.org/en/about/.