# Python Proxy Herd Implementation using ‘asyncio’

Adam Cole
CS 131 Programming Languages - Winter 2020

## Abstract

In this project, our goal was to implement Server and Client routines which would be used to create nodes of a proxy herd in multiple terminal windows. Since Python does not have built-in support for asynchronous execution and multithreading, we implemented our server heard with the ‘asyncio’ python library. Using ‘asyncio’, we are able to run python code asynchronously and are able to maintain multiple client server connections concurrently. We also investigated the similarities and differences between asynchronous technologies, Node.js and Python’s ‘asyncio’, as well as the similarities and differences between Python and Java as programming languages.

## 1. Introduction

Wikipedia and its related sites are based on the Wikimedia Server platform, which is based on GNU/Linux, Apache, MariaDB, and PHP+JavaScript, using multiple, redundant web servers behind a load-balancing virtual router and caching proxy servers for reliability and performance. This is a typical method of launching and supporting web applications, but drops in performance with increasing requests to update articles.

We want to develop a new Wikimedia-style service designed for news, where (1) updates to articles will happen far more often, (2) access will be required via various protocols, not just HTTP, and (3) clients will tend to be more mobile. Clearly, the Javascript and PHP will create a bottleneck in our construction. Our goal is to investigate a different server architecture called an ‘application server herd’, where multiple application servers communicate directly to each other as well as via the core database and caches. We will use this architecture to support rapidly-evolving data and bypass accessing slow databases.

To investigate the use of application server herds for our platform, we chose to develop a simple and parallelizable proxy for Google Places API using Python’s ‘asyncio’ library. Our boss is worried, however, about Python’s implementation of type checking, memory management, and multithreading may cause problems for large scale use.

## 2. Python ‘asyncio’ Library

As mentioned earlier, Python lacks a built in ability to support concurrency. Python’s ‘asyncio’ library allows Python developers to write concurrent code to be executed asynchronously. Furthermore, ‘asyncio’ allows applications to support multiple connections concurrently, which are vital to many functions. This can help Python developers create web servers, database connections, and distributed tasks. While ‘asyncio’ effectively provides these APIs to Python developers, it is designed to run on a single thread only.

### 2.1 Syntax

Including the ‘asyncio’ library adds two keywords to the Python syntax, ‘await’ and ‘async’.

The ‘async’ keyword must be used just prior to function definitions, which specifies that the function can be run asynchronously as a coroutine. Adding this keyword changes how the coroutine function must be called, too. In order to call a coroutine, the function must be added to a task on an event loop, or called with the ‘await’ keyword. After calling this, the function is run concurrently with the code that follows sequentially.

The ‘await’ keyword is used to call asynchronous coroutines. This keyword can also run any awaitable object, which coroutines are a subset of. If we try to call an asynchronous function without ‘await’, Python will create a coroutine object but not run it, so coroutines must be called with ‘await’.

### 2.2 Main Features of ‘asyncio’

Many uses of the ‘asyncio’ library have been mentioned already, but here we list out the most important implementation features of the library. First, the ‘asyncio’ library uses an event loop to execute the running program, and serves as a “program” operating system - scheduling tasks and chunks of code to execute and when.

Secondly, there are two ways of implementing asynchronous code. 1) Coroutines, which are defined with the ‘async’ keyword in front of function definitions and called with the ‘await’ keyword. 2) Tasks, which are logically similar to coroutines but must be used with the program’s event loop. By scheduling tasks, we can add tasks to be run in the future.

### 2.3 Advantages of ‘asyncio’

One of the biggest advantages of ‘asyncio’ we have already discussed in depth - its ability to abstract the concurrency process for Python developers. This not only makes it easy to implement concurrency on programmers, but also boosts performance for HTTP requests or other remote queries.

The next biggest advantage of ‘asyncio’ is the library’s ability to support multiple connections concurrently. Combining these two advantages allows Python to become a much more powerful language than before.

### 2.4 Disadvantages of ‘asyncio’

The ‘asyncio’ library poses two main disadvantages. First, while ‘asyncio’ takes great strides to achieve concurrency in Python - a strictly imperative programming language - it will never be able to achieve total parallelism. Python does not support multithreading, and neither does the ‘asyncio’ library.

Second, the increased complexity of programs that ‘asyncio’ can help create raises debugging issues. Since we can’t always be sure what order the code was executed in, debugging programs can be tougher and have inconsistent results. While this is true of all concurrent programs, it raises a consideration to be made when comparing to Javascript + PHP.

### 2.5 Server Herds and ‘asyncio’

The ‘asyncio’ library works perfectly to implement a server herd. To recap, a server herd is where multiple application servers communicate directly to each other instead of reading back to a common database / resource. The ‘asyncio’ library functions were developed exactly for a purpose like server herds - easily abstracting processes that open connections to handle multiple connections and allows for concurrency and multiple requests. Next, we describe how we implemented our server herd for parallelizable proxies for the Google Places’ API.

## 3. Implementing Server Herds using ‘asyncio’

The plan of attack we used to implement our server herd in python was through a Server Class. In this way, we approached the problem

with an Object-Oriented style. The class variables were organized in multiple python data structures. First, a list of class variables and their names (Hill, Jaquez, Smith, Campbell, and Singleton) were kept. Each of these variables represented a local server that we would deploy and add to our server herd. A Python Dictionary was also used to store the connections between servers. I have the connections displayed here:

| SERVER | CONNECTIONS |
|---|---|
| HILL | JAQUEZ, SMITH |
| JAQUEZ | HILL, SINGLETON |
| SMITH | HILL, CAMPBELL, SINGLETON |
| CAMPBELL | SMITH, SINGLETON |
| SINGLETON | JAQUEZ, SMITH, CAMPBELL |

When run, the server object retrieves the event loop from the 'asyncio' library and continuously runs 'handle_client' when a TCP connection is established with another process (be it a client or neighboring server). We set this function to be an asynchronous coroutine in order to allow multiple connections and ultimately our server herd to function. If no coroutine was used, one connection would have to wait for another connection's process to terminate.

In order to implement our Google Places proxy, we allowed three different types of messages to propagate through our server herd. First, an IAMAT message from clients connected to our server, which would communicate the client location. Since we sought to implement a server herd, we then must communicate this to all other servers. This was done by 'flooding' the client information to all local servers. We implemented flooding through a coroutine which forwarded the client information our second message type - an AT message - to all connected servers. These servers, when receiving the AT message, update the client's information and forwards to all of its connected servers. In this way, our servers can keep updating quickly-changing information, like a client's GPS location.

The third type of message, a WHATSAT message, is a client message that HTTP queries the Google Places API about a nearby location. While IAMAT and AT messages are flooded through the server herd, WHATSAT messages are not since they do not update any client information. It was important to create this message handler as a coroutine since HTTP requests have large latencies compared to the rest of our program's operations.

To put our server herd to the test, we ran:

```
$ python3 server.py ServerName
```

on five different terminals, each with a unique server name. By running all five servers at the same time, we create our server herd. We create a test client by opening another single terminal session and run:

```
$ python3 client.py
```

and sent our servers IAMAT messages to propagate flood AT messages containing client information through the server herd.

## 4. Analysis of Implementation

After implementing the server herd, 'asyncio' performed well under the specified conditions we placed around the framework. It was difficult to test the project before almost complete development, and even after testing began, finding errors was not as difficult as predicted. Our server herd performed well and would be a beneficial option to consider when developing our new Wikimedia-style service.

### 4.1 Java and Python Comparison

Instead of using Python, another implementation option would be to use Java, a programming language known for easily implementing web applications and supporting concurrency well. Both Python and Java have their pros and cons, so next we explore the two languages in terms

of memory management, type checking, and multithreading - the categories our boss was worried about for Python.

### 4.1.1 Memory Management

Both Python and Java implement their programs with stack frames and a heap of free memory. The main difference between the two languages is how they recollect unused memory - commonly known as garbage collection.

Java collects ‘garbage’ by tracing objects through the program execution tree. The point of tracing is to determine which allocated objects are still referenceable by the program, and will garbage collect those objects that are no longer referenceable. This method is called ‘Mark and Sweep’, where objects are ‘marked’ if they are no longer referenceable, and then ‘swept’ up and recollected. Tracing objects as Java does is more robust than Python’s garbage collection, but is less efficient since it must iterate through all objects in order to mark those no longer referenceable.

Python garbage collects through reference counting. When developers create objects, Python inherently allocates a few bytes before the object in memory and keeps track of how many variables reference the object currently. In this way, Python garbage collects all objects with reference count == 0. While this is much more efficient than Java’s mark and sweep, it is less robust due to the possibility of circular referencing. If two structures reference each other, but neither structure is referenceable by the executing program, the two objects will never be garbage collected even though they should be.

### 4.1.2 Type Checking

Java and Python are both strongly typed languages. This indicates that implicit type conversions are not done by the language when assigning and executing. Python and Java differ, however, in the manner of its type checking. Python is a dynamically typed language, while Java is statically typed. Statically typed languages check all types prior to execution, while Dynamically typed languages type check during runtime.

Understanding this, we conclude Python will be easier to develop, but will be more prone to runtime errors and have slower run times since it type checks during execution. Oppositely, Java will take longer to develop due to type restrictions during compilation, but will run faster since no type checking occurs during execution.

### 4.1.3 Multithreading

Java and Python differ the most in this category, since Python is not designed to support multithreading even with ‘asyncio’ and other useful available libraries. Python inherently ensures only a single thread can be executing the program at any given time. Java, on the other hand, was defined for concurrency at all levels. For example, there are built in Java keywords for synchronization, a necessary feature for multithreading and avoiding Data Race Conditions (DRC’s).

## 4.2 Node.js and ‘asyncio’ Comparison

The comparison of Node.js and ‘asyncio’ acts as an extension of our comparison of Java and Python as programming languages. The Node.js framework helps implement Javascript applications, while ‘asyncio’ helps implement Python applications. Both Node.js and ‘asyncio’ serve as asynchronous event-driven execution tools - and therefore are both options to implement our server herd prototype.

A key difference between Node.js and ‘asyncio’ is that Javascript’s Node.js was designed top - bottom to support asynchronous code, while Python was not and ‘asyncio’ works to solve this issue. In this way, Node.js will run faster than Python’s interpreter with the ‘asyncio’ library.

Another notable difference between the two frameworks is the learning curve. Python is commonly considered the easiest language to pick up (even if it's your first language), and Node.js is a specific platform that would require training and extra resources than Python would.

## 5. Conclusion

Python is a simple language with a very widespread user base and extensive support, and can easily implement a server herd. Python gets the job done, but does not optimize memory management nor performance. Furthermore, since 'asyncio' is a Python library, future versions of Python could affect 'asyncio' execution. Due to this, Python would be a great language to test a model for our server herd architecture. For deployment, however, Node.js should be used in Java and Javascript to implement the final design. Node.js and Java, while development would take longer, would support larger applications with varying scale much more robustly than Python. And in the end, robustness and performance are the most important components of a service - not the required development time.